

Table of contents

Table of contents	1
Analysis	4
Description	4
Target Audience/Stakeholders	4
Why a computer is suitable for the task	4
Research	5
Features and limitations	6
Requirements	7
Success Criteria	7
Design	8
Decomposition of the problem	8
Problem 1.1 – How is the ‘data from past matches’ stored?	8
Problem 1.2 – What data is actually stored?	9
Problem 1.3 – How is the state of the board stored?	10
Problem 1.4 – A hash of a board is only valid if the AI only plays the same piece.	11
Problem 2.1 – How does the AI use this data to decide a move?	12
Problem 2.2 – How does the AI determine a ‘path’ (also, how is the win percent calculated)? .	13
Problem 2.3 – How does the AI decide which path has the highest win percentage?	17
Problem 3.1 – What does the AI do during a match?	18
Problem 3.2 – How does the AI figure out what move to make?	19
Problem 3.3 – What algorithms does the AI use to handle these problems?	20
Proposed structure of the program	21
My reasons for proposing a synchronous (multi-threaded) approach	21
Communication between threads	21
The Game loop	23
The Game Thread	26
The GUI Thread	27
The AI Controller	27
[OnStart]	27
[OnDoMove]	27
[OnAfterMove]	27
[OnEnd]	27
The Player Controller	28
[OnStart]	28

[OnDoMove]	28
[OnAfterMove]	28
[OnEnd]	28
Usability	28
Test Data for development	30
Test Data for beta testing	32
Development.....	33
Iterations of development, prototypes, and testing	33
Evidence of modular code	33
Evidence of validation.....	37
Review.....	41
Evaluation	42
Testing and success criteria	42
Usability Features	45
Overall evaluation.....	53
Future Maintenance	53
Roadmap	55
V0.1.0	56
Final version for V0.0.X	56
V0.0.1	57
V0.0.2	58
V0.0.3	59
V0.0.4	60
V0.2.0	61
V0.1.1	63
V0.1.2	65
V0.1.3	68
V0.1.4	71
V0.3.0/V1.0.0	74
V0.2.1	75
V0.2.2	76
V0.2.3	77
Chronological Order of Development.....	79
Code Appendix.....	82
Board.cs.....	82
Config.cs.....	97

GameFiles.cs	98
ISerialisable.cs	103
Message.cs	104
Node.cs	105
Controllers/Controller.cs	116
Controllers/AI.cs	118
Controllers/PlayerGUI.cs	125
NodeDebugWindow.xaml	128
MainWindow.xaml	129
NodeDebugWindow.xaml.cs	132
MainWindow.xaml.cs	135
Unittests/GameHash.cs	144
Unittests/AverageTests.cs	147
Unittests/BoardTests.cs	148
Unittests/GameFilesTests.cs	150
Unittests/NodeTests.cs	153
Pseudocode/CalculateAverages.txt	157
Docs/Overview.docx	158
Overview	158
[Details] How will the AI work?	158
How the game board is hashed	158
How is the data from each match stored?	159
How will the AI behave during a match?	160
During a match [Technical]	161
At the end of a match [Technical]	164

Analysis

Description

The aim of this project is to create a virtual version of the game Tic-Tac-Toe with an AI that learns how to play using data from its previous matches.

Tic-tac-toe is a 2-player game played on a 3x3 grid. One player will play as the 'X' piece, while the other player is the 'O' piece. Each player takes a turn to place their piece on the grid. The aim of the game is for one of the players to get 3 of their pieces next to each other in a row; which can be done vertically, horizontally, or diagonally.

The AI will be used to substitute one of the required players, and the aim is for this AI to learn from its previous experiences to try to figure out what is a likely way of it being able to win.

The AI should look at its past history of matches, and use this data to figure out the best way to continue during a match. However, the AI won't always have enough data to use (it learns while it plays games, so if it hasn't played many games, there's not a lot of data) so it should also have a way to deal with such a situation, such as doing a move completely at random. The AI could then use this new data in future matches, giving it more variety in how it can play.

Target Audience/Stakeholders

There is no specific audience due to the simple/casual nature of the game, and the fact that most people seem to know what tic-tac-toe is means it is very accessible to a large number of people. This means that the stakeholders will most likely be anyone who is aged 6 years or older.

The game should provide instructions however, in the case someone does not know how to play tic-tac-toe.

There is no need for technical expertise when playing the game, so it is accessible to most people.

Why a computer is suitable for the task

Computers are very fast at performing calculations, and the only errors they make are generally due to human errors (coding mistakes, or errors intentionally put in, for example). For a game as simple as tic-tac-toe, a computer is more than capable of calculating what it should do in a reasonable amount of time, and can possibly be almost impossible to beat.

As an example, when two human players are playing against each other, Player 1 may be one move away from winning the match with Player 2 not noticing they can stop them; however, when a human is against an AI, the AI can be made so it will always block the human from winning, if there's a chance to. This is due to the point made earlier, the only errors a computer can make are usually due to human errors, so if a human tells the computer to always (or to never/only sometimes) block the other player, then it will do so without fail (assuming there's no errors in the code).

As another example, for a simple game like tic-tac-toe, the computer may be able to plan ahead of time and think of the most optimal route to take, similar to a human. The difference is that a computer can analyse the possible paths it can take significantly faster than a human, and a computer will be able to 'remember' them all perfectly, whereas a human might forget something or make mistakes in their logic.

Computers are also capable of storing tremendous amounts of data. For this project's use case, this is good as it allows the game to be able to store data of hundreds or thousands of unique tic-tac-toe matches for use with the AI.

Research

While researching on what algorithms I might use when writing the AI for the game, I came upon the Minimax^[1] algorithm.

The Minimax algorithm as defined on Wikipedia is...

A decision rule used in decision theory, game theory, statistics, and philosophy for minimizing the possible loss for a worst case (maximum loss) scenario.

After further research, I came upon a website^[2] where a programmer describes how they used the Minimax algorithm with tic-tac-toe. The general idea is, they calculated every possible route the AI could take, and awarded points to each route which signifies how much of a loss (-10 points) the AI would suffer if it went down this path, and how much of a gain (+10 points) it would get. The path with the highest amount of points would be chosen. (The website also talks about other tweaks needed to make it work well with tic-tac-toe).

The issue with this algorithm is, it creates an unbeatable AI, which is not fun for the human to fight against (nor does it seem terribly interesting to code). The upside is, this algorithm is a perfect example of how a computer is suitable for playing tic-tac-toe, and can be better at it than humans.

The idea of weighing which path is most likely to win/lose was interesting to me, and during a session with my computer science tutor, he was discussing about possibly using machine learning, where the computer stores data of past games and then uses that data to determine which moves have led it to a win in the past.

The advantage of the AI using past data, is that instead of calculating the best moves to make on the spot, is that it can attempt to 'learn' the best way to win which I see as an acceptable compromise between 'impossible to beat' and 'impossible to lose against'. At the start, when the AI lacks data, it should be pretty easy to beat; but as time goes on the AI will 'harden' and gradually get more data meaning it will be able to perform better than when it started.

Similar to how the minimax algorithm would create a tree of moves to analyse, my AI could store the data of its past games in a tree. For example, it may be formatted like:

-> "X is placed in the top-middle slot" -> "O is placed in the bottom-right slot"
"empty grid"
-> "X is placed in the top-left slot" -> "O is placed in the bottom-middle slot" etc.

[1] <https://en.wikipedia.org/wiki/Minimax>

[2] <http://neverstopbuilding.com/minimax>

Features and limitations

The game must provide a GUI. This GUI must display the 3x3 grid which shows the current up-to-date state of the match. The GUI must at the very least allow the player to play multiple matches without having to restart the game. Finally, the GUI must allow the player to interact with the 3x3 grid, following the rules of how you're allowed to play pieces in tic-tac-toe.

The game should provide a message box that details how to play tic-tac-toe. Ideally this should be shown when the game is opened for the first time, and whenever the user presses some sort of "help" button.

The game must not allow the user to perform an invalid move, and should simply wait for the user to input another move if this happens.

The game will require having to store data on previous games, and being able to load this data when it is opened. The game should use a binary format, as it allows for more compact file sizes, but there is a trade-off of a human (me in particular) being able to easily read and debug the data as would be possible using a text format.

Multiplayer, while a desirable feature, is not the focus point of the project; that would be the AI. Therefore, multiplayer capabilities won't be added to the game until sometime in the far future, if at all.

The game will require an AI for a human to fight against. This AI should make use of its past matches with humans to aid it with choosing what moves to make during a match. The AI should not be unbeatable, as it would be unfun to fight against.

An animated GUI that comes with sound effects is quite a bit of effort with very little worth considering how simple a game tic-tac-toe is, so I have decided to go with a very simple, soundless GUI.

Due to the reason that the AI learns as it plays, it will start off being incredibly easy to beat, but over time it will become more challenging. Theoretically, it should only end up either winning or tying after a while (something I wish to avoid, due to it being unfun); however, this will require over two-hundred thousand unique games to have been played^[1]. Because of this, it may take some time for the AI to actually be a considerable threat, and is likely to be very easy to beat for longer than I'd like it to be.

[1] <https://www.jesperjuul.net/ludologist/255168-ways-of-playing-tic-tac-toe>

Requirements

OS: Windows Vista SP2 (with .Net 4.5 installed) or later (Any Windows OS that can run WPF ^[1])

CPU: 2GHz or faster.

GPU: Integrated graphics card, or better.

The project will be coded in C#. It will use the WPF framework for the GUI. This means that a Windows OS must be used, as WPF (and C#, for the most part) is only supported on Windows.

The project will be built and tested against .Net 4.5, so .Net 4.5 must be installed on the computer. The project *might* work with older versions of .Net, but it is not guaranteed. .Net 4.5 comes preinstalled with Windows 8 and later versions of the Windows operating system.

There is no reason in particular as to why .Net 4.5 was chosen, so it is most likely possible to compile the project for an older version.

There are no special graphical requirements for the project, so any GPU that can comfortably run Windows is all that is needed, graphics wise. While any CPU will also work fine as well, a CPU that is *too* slow could make the AI begin to take seconds before it performs the move, possibly making the game feel bad to play.

[1] https://en.wikipedia.org/wiki/Windows_Presentation_Foundation

Success Criteria

To be deemed a success, the game must provide:

- A user-friendly, responsive GUI that provides: the 3x3 grid with an up-to-date view of the game board's state; text informing the player which piece they're playing as; text that displays whether it is the player's or AI's turn, and it must allow the player to place their piece via the 3x3 grid.
- An AI that is not impossible to win against, and is capable of analysing the data from its past matches to determine which move it should take.
- The game must not crash unexpectedly, and in the event that something goes wrong, it must simply show the user an error box saying something's gone wrong.
- The game must be stable and free of any major bugs (for example, if the GUI suddenly stopped functioning, this is a major bug and should not happen). Certain features of the game, and small parts of the code can and should be tested. The preferred method of testing is unit testing, where a small piece of code is written to test a very specific part of the code. Features of the game that are tricky to test via code (such as how the GUI functions) should be manually tested and documented.

Design

Decomposition of the problem

The problem as a whole for the AI can be described as 'An AI using data from past matches to determine which moves will most likely result in a win.'

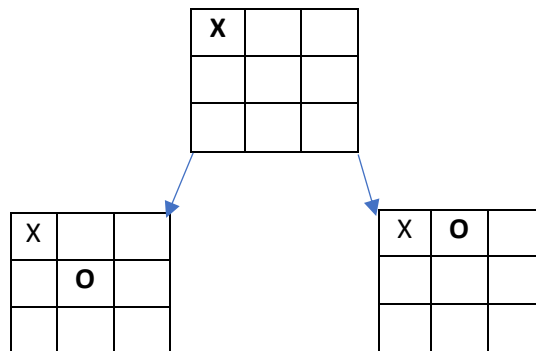
Problem 1.1 – How is the 'data from past matches' stored?

The AI requires a way to store and use data of multiple tic-tac-toe matches.

A tree would be a suitable data structure to use due to the nature of tic-tac-toe. For example, a node in the tree might describe the following state of a tic-tac-toe board, after the AI (for example) sets their piece at the top-left slot:

X		

The player (again, as an example) could then place their piece in any of the other empty slots:



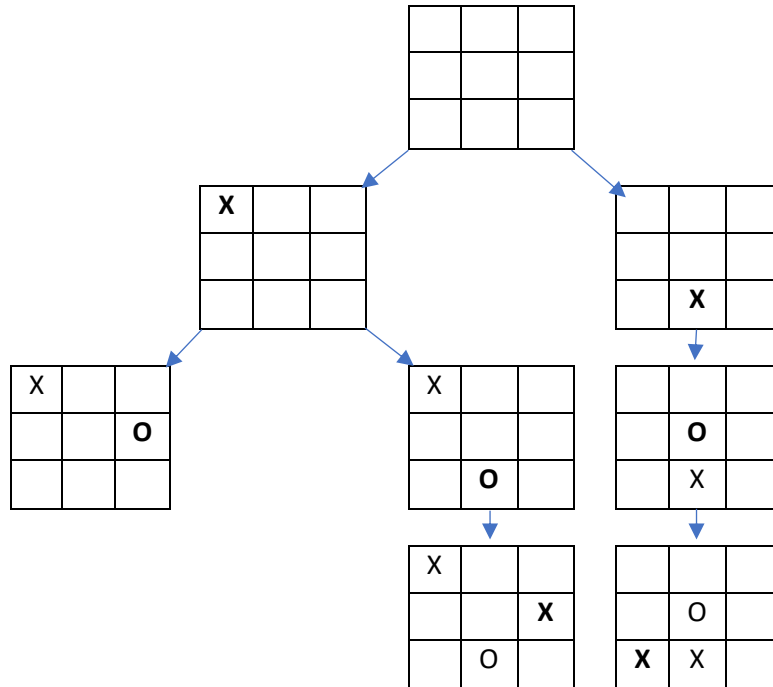
As shown in the diagram, the bottom-left node represents when the player places their piece in the middle slot, **after** the AI places its piece at the top-left slot; while the bottom-right node represents when the player places their piece in the top-middle slot, also **after** the AI places its piece at the top-left slot.

As the diagram demonstrates, a tree would be a very natural data structure to represent data from numerous tic-tac-toe matches, as it would support being able to store data about every possible match of tic-tac-toe, and stores it in a logical manner.

"AI places X in the top-left slot" -> "Player places O in the top-middle slot" -> etc. is very easy to represent in a tree (as long as the nodes can have any number of children).

Here is an example of a slightly more fleshed out tree to further demonstrate the viability of using a tree for this data. The root of the tree is simply an empty board. The move that each node in the tree represents has been emboldened:

[Figure: Example tree]



Problem 1.2 – What data is actually stored?

Each node in the tree needs to store data that allows the AI to determine whether it is likely to win or not.

My proposed solution would be for a node to store the following:

- The state of the game board, representing a single move (See Problem 1.1).
- How many times the move the node represents has led to the AI winning.
- How many times the move has led to the AI losing.
- The index of which slot (range of [0..8] inclusive) on the game board the move was performed on. For example, the index of 0 means that the move the node represents placed its piece in the top-left slot, whereas an index of 8 means that the piece was placed in the bottom-right slot.

A notable thing to point out is that, while it **is** possible to determine the index of where a piece was placed by looking at the previous node's board state, and comparing it with the current node's board state, it will be more simple (and more importantly, less buggy/more stable) if each node simply stored the index.

Storing how many times a move has caused the AI to win/lose allows the AI to calculate the win percentage of a node – what percentage of games it has led to the AI winning. The formula for calculating this is $\text{win}\% = \frac{w}{(w+l)}$ where 'w' is the number of wins, and 'l' is the number of losses. This percentage can be used by the AI's algorithm to determine what move to perform.

Storing the index of where the piece was placed is done so the AI can replicate the move during a match. For example, if it picks a node where the index is 2 (the top-right corner), then the AI will know that it should place its piece at slot 2 to replicate the move that the node represents.

Problem 1.3 – How is the state of the board stored?

Now that I have specified how the data will be stored (in a tree), and what data the nodes store, I now need to determine what kind of ‘format’ the data in a node is stored in.

For the win counter, loss counter, and slot index stored in a node, it is quite clear that they are numbers. However, nodes must also store the state of the game board which so far, have not been provided a ‘computer-friendly’ way to be represented.

The solution is pretty simple, a 9-character string is stored with the node, where the 0th character represents the 0th slot of the game board, the 1st character represents the 1st slot, etc. I refer to this as a *hash* of a game board.

For example, take the following game board:

X		
	O	
	O	X

The hash of it would be “X...O..OX”, where an ‘X’ represents the X piece, a ‘.’ represents an empty space, and an ‘O’ represents the O piece.

Problem 1.4 – A hash of a board is only valid if the AI only plays the same piece.

This solution has a slight flaw however, if the AI was **always** the X piece, and the player was **always** the O piece, then this solution would be fine, but if the AI were to suddenly become the O piece, and the player suddenly became the X piece, then the data would no longer be valid because, while the AI and player have changed which piece they've used, the hash itself doesn't reflect these changes.

This problem has an easy fix; instead of storing 'X' to represent the X piece, and 'O' to represent the O piece in a hash, we instead store 'M' to represent the AI's piece ('M' stands for 'Mine'), and 'O' to represent the player's piece ('O' stands for 'Other player').

As an example of this new idea, take the previous game board; If the AI is X and the player is O, then the hash would now become "M...O..OM". Now, if the AI was O, and the player was X, then the hash would become "O...M..MO".

The two tables below demonstrate what the hash "O...M..MO" would look like if the AI was playing as X (represented by the left table), and if the AI was playing as O (represented by the right table).

O		
	X	
	X	O

X		
	O	
	O	X

This means that a node with a board hash of "O...M..MO" would be useable regardless of if the AI was playing as the X piece or the O piece, whereas the old idea ('X' for X, 'O' for O) would make the nodes incompatible depending on which piece the AI plays as.

When the hash needs to be converted back into a board state, then 'M' can simply be replaced with what piece the AI is using, and 'O' is replaced with whatever piece the player is using.

It's worth noting that, even if it's unlikely for the AI to be able to change which piece it uses in my project, I still find it important that the data is reusable (there is no difference between which piece the AI uses, so the data should be the same for whether it's playing as X or O), which is why I came to this solution. It will likely require little effort to implement while providing a rather large bonus, making it worthwhile.

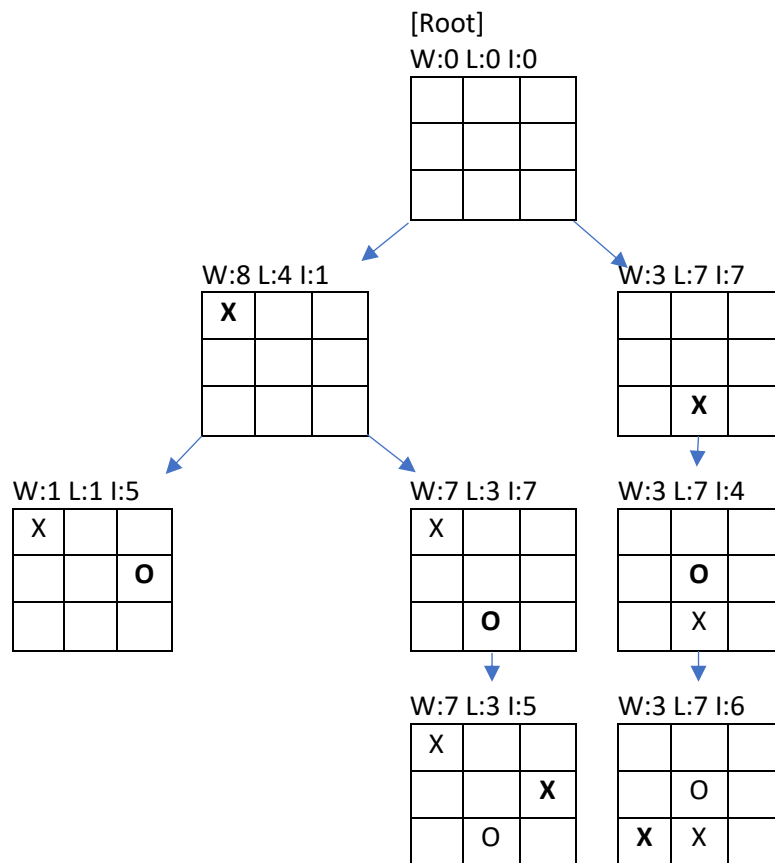
Problem 2.1 – How does the AI use this data to decide a move?

At a high-level, all the AI needs to do is go over every path in a tree of moves it has created from its past matches, and select the path that has the highest win rate.

Here is the example tree displayed in Problem 1.1, but with the extra data explained in Problem 1.2 (Excluding the hash, for readability). ‘W’ means ‘wins’, ‘L’ means ‘losses’, and ‘I’ mean ‘index’. The AI is ‘X’, the player is ‘O’. This tree will be used in later examples.

The root node has the value ‘0’ for the wins, losses, and index. This is because the root does not represent an actual move, so should not have these values modified.

[Figure: Detailed Example Tree]



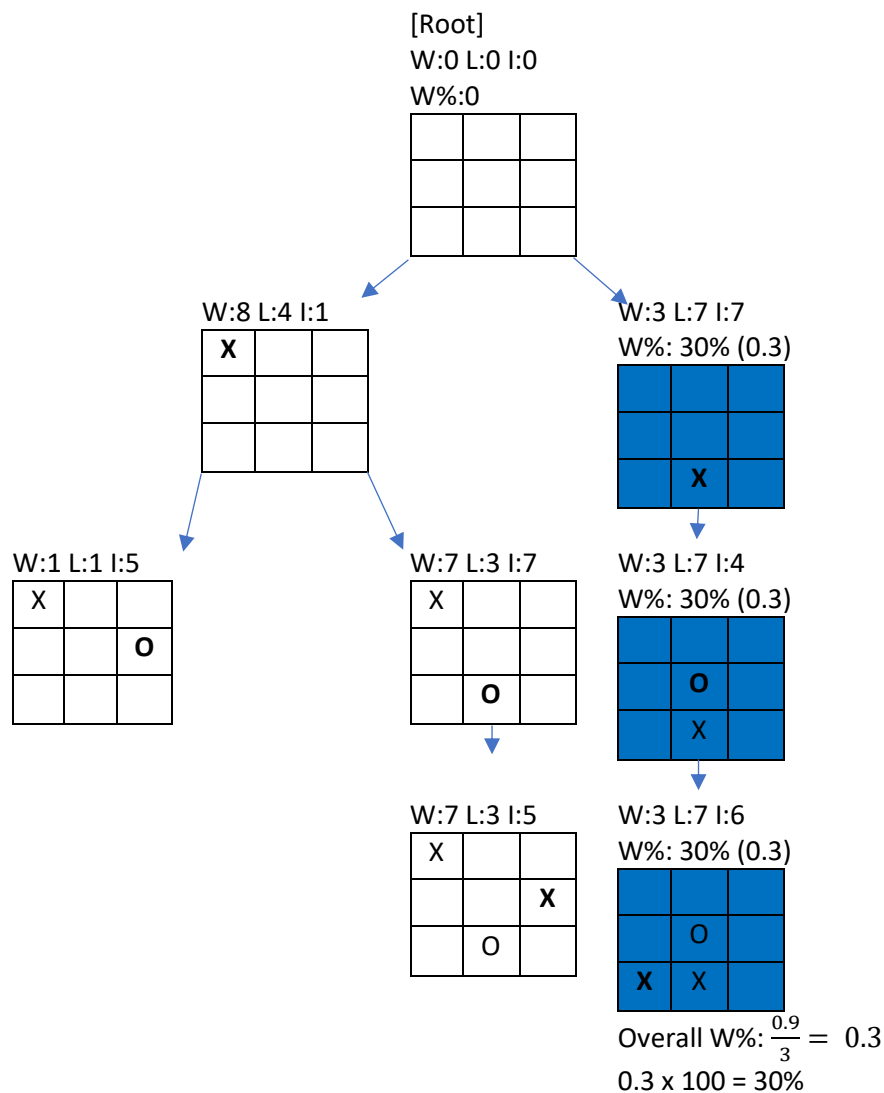
Problem 2.2 – How does the AI determine a ‘path’ (also, how is the win percent calculated)?

A path is a list of nodes that make up a tic-tac-toe match. I will provide a high-level example of the paths that exist in the example tree shown in Problem 2.1, then describe the algorithm that can be used to determine all the paths.

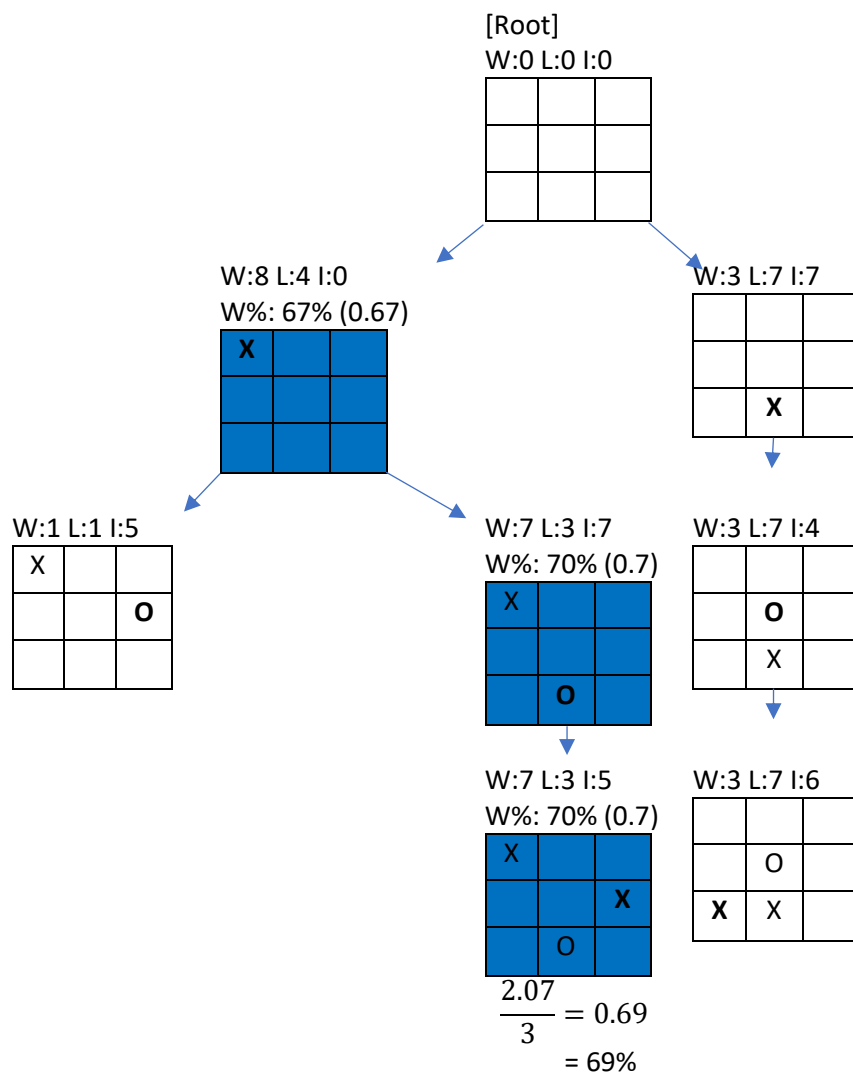
The diagrams below show all of the different possible paths the AI could take, where the nodes coloured in blue are the nodes making up the path. The win percentage (depicted as ‘W%’) of each path is calculated (using the formula in Problem 1.2), and the average of the win percentages is used to determine the overall likelihood of the path winning. The win percentages are calculated in this section so later sections of the document can reference to it.

To find the average win percent, add up all of the win percentages of the nodes that make up the path, then divide the number by how many nodes are in the path. If the percentages are used in decimal form (60% being 0.6, 25% being 0.25, etc.) then multiply the result by 100 to get a more ‘natural’ percentage (e.g. $0.25 \times 100 = 25\%$).

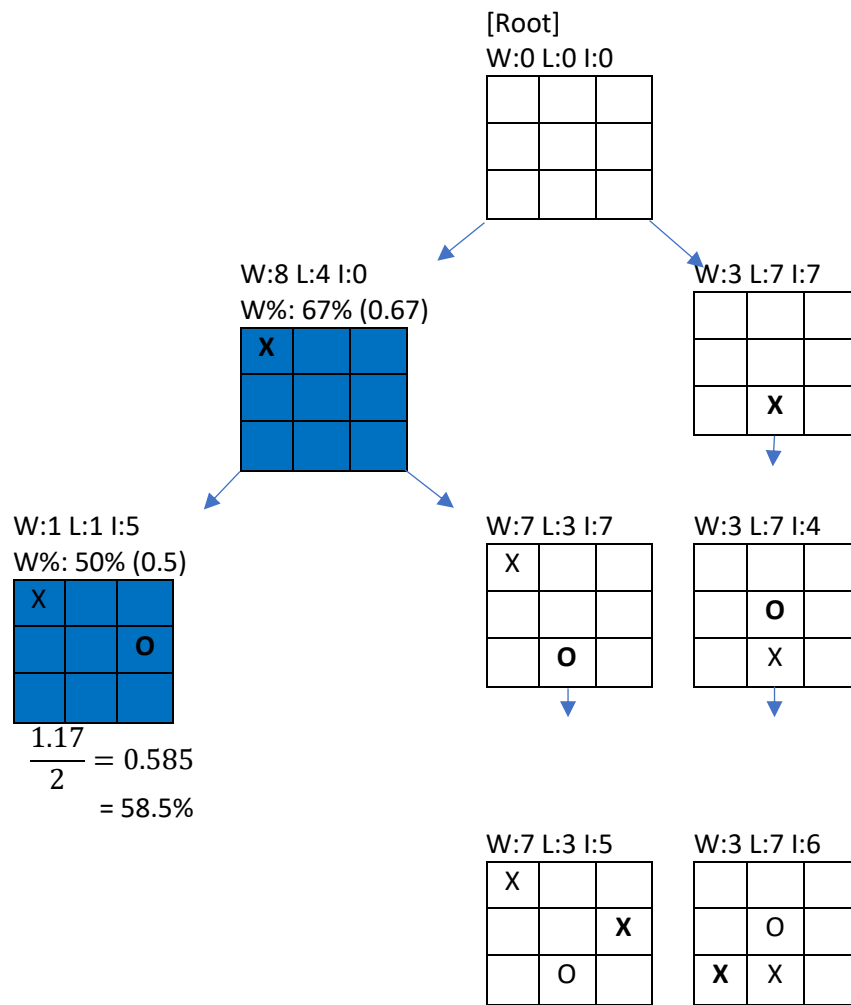
[Figure: Path #1]



[Figure: Path #2]



[Figure: Path #3]



This is the algorithm used to find the paths in a tree. This algorithm can be used to simplify other algorithms, and will be referred to as the 'WalkPaths' algorithm in this document. It may also be worth noting that WalkPaths is recursive, so the algorithm uses itself in a 'divide-and-conquer' fashion. This is an $O(n)$ algorithm, as the amount of time it takes is related to how many nodes there are in the tree.

The algorithm at a high level:

1. Starting with the root, get the next child node, and add it to a list.
2. If the child node also has children, perform step 1 on it.
3. If the child node does not have children, then it is the end of a path, so perform an *Action*(defined below) on it.

Variables:

- **Node** = The node passed to the first step in the algorithm. The first node given is defined as the 'Root' node.
- **Path** = An array of nodes.
- **Action** = Another algorithm that is performed on **Path**.

WalkPaths is defined as:

1. If **Node** isn't the root node, add it to the end of **Path**. *
2. If **Node** doesn't contain any children nodes, then it's the end of a path, so,
 - a. Perform **Action** on **Path**.
 - b. Go to Step 4.
3. Otherwise, get the **Node's** next child.
 - a. Perform WalkPaths, where **Node** is now the child, **Path** is the same **Path** currently being used, and **Action** is the same (come back to this step afterwards).
 - b. Remove the last node in the **Path** (this is the child node we just got, and would've been added by step 1).
 - c. Perform Step 3 again if the **Node** still has children to go over.
4. If **Node** is currently the root node, then the algorithm has gone through all paths, so end the algorithm.
5. Otherwise, set **Node** back to its previous value.

* *The Root node doesn't represent a move, so shouldn't be included in the Path.*

I recommend using the WalkPaths algorithm on the example tree (*Action* can simply be defined as 'do nothing' in this case) if it is unclear how it works.

The *Action* is task-specific, so should default to 'do nothing' if no specific action is defined. Later sections will specify an *Action* to demonstrate how generic and reusable this algorithm is.

Problem 2.3 – How does the AI decide which path has the highest win percentage?

Now that we have defined the 'WalkPaths' algorithm, and it is made in such a way that we can attach an 'Action' for it to perform on every path, it makes other algorithms very simple - such as the algorithm the AI will use to determine which path has the highest win percentage.

This is an $O(n)$ algorithm, as the amount of time it takes is in proportion to how many nodes are in the *Paths* given to it. When used together with the WalkPaths algorithm, they become an $O(2n)$ algorithm, as they go over every node in the tree twice.

I will clarify that the two algorithms, when put together, will actually be **slower** than $O(2n)$ since the amount of times a node is walked over is actually related to how many different paths the node can lead to, but my poor knowledge of defining algorithms with Big-O notation prevents me from providing the correct notation.

The algorithm at a high level is simply to:

1. Go over every path in the tree.
2. Pick the path with the highest average win percentage.

To do this, the AI will perform the WalkPaths algorithm, where 'Action' is defined as:

Variables:

- **BestPath** = The path (array of nodes) that has the highest chance of winning.
- **BestPercentage** = The win percentage of **BestPath**.
- **Path** = The path given to this algorithm by the WalkPaths algorithm.
- **Percentage** = The overall percentage of **Path**. This value is the percentage as a decimal (25% being 0.25 as a decimal, for example).

Steps:

1. Get the next node in **Path**.
2. Calculate the win percentage (described earlier in the document) and add it to **Percentage**.
3. If there are still nodes in **Path**, go to step 1.
4. If **Percentage** is greater than **BestPercentage**.
 - a. Set **BestPercentage** to **Percentage**.
 - b. Set **BestPath** as a copy of **Path**.

At the end of the WalkPaths algorithm, the **BestPath** variable will hold the path with the highest average win percentage. As before, I recommend to try it out on the example tree (as the example diagrams show, the result should be the same as 'Path #2').

This algorithm also demonstrates the reusability of the WalkPaths algorithm, as WalkPaths handles finding every path in a tree, while the 'Action' algorithm can focus purely on the task its designed for.

I generally refer to this algorithm as the "StatisticallyBest" algorithm, and will use this name to reference it later on.

Problem 3.1 – What does the AI do during a match?

Now that we know how the data is stored, and have some algorithms to use on this data, we now need to figure out what the AI should be doing during a match.

The AI first of all, needs to keep a tree full of nodes from past matches; this is called the 'Global' tree. This Global tree, because of it holding all of the AI's past matches, is the prime candidate for the data the AI will be using to decide it's move.

The AI should also keep another tree which represents the current match it is playing, and is called the 'Local' tree. This tree is needed so the AI can 'mirror' itself from the Local tree into the Global tree.

So overall:

- After every move (whether it's by the player or the AI) create a node for the move, and add it into the local tree.
- During the AI's turn, it should use the data in its Global tree to figure out what move to make.
- At the end of a match, the AI should bump the 'win' and 'loss' counter of each node in its Local tree, and then merge it into the Global tree.
- Either after a match, or when the game is closed, the AI should save its Global tree into a file.
- At the start of a match, the AI should load its Global tree from a file (if one exists), and empty its local tree.

Problem 3.2 – How does the AI figure out what move to make?

It's simple to say to myself 'Just use the StatisticallyBest algorithm and that's that' but unfortunately there are some problems that must be solved.

Imagine the AI is in a match, and its Global tree is the example tree shown earlier (the 'Detailed Example Tree' in Problem 2.1). Now, if the WalkPaths algorithm is used, where its *Action* is the StatisticallyBest algorithm, then the AI would choose to go down 'Path #2' (from Problem 2.2).

So, the AI has chosen Path #2, and starts off the match putting its piece in slot 0 (since the first node in the path is 'AI puts piece in slot 0', meaning the AI will mimic it). Now, there are a few things that can happen depending on what the player decides to do.

If the player places their piece in slot 7 as the second move (the second node in Path #2 represents this move) then the AI can keep using the path it selected, so it will put its piece in slot 5 (the last node in Path #2). This scenario is the easiest to handle, since the AI will be able to walk down the path it chose at the start of the match. While not a terribly clear name, this is the 'Matches selected path' scenario.

If the player places their piece in slot 5 as the second move however (take a look at Path #3, the second node in that path represents this move) then the match has gone off track from the path the AI selected (Path #2), but the Global tree has a node for the move the player has chosen (in Path #3), so it could re-do the WalkPaths algorithm where the node representing the move the player did (the one that caused it to go off-track from the previous path) is used as the root *Node* parameter. In short, if the player performs a move that goes off track from the selected path, and if there's a node in the Global tree for this move, then recalculate the StatisticallyBest path using the node for the player's move as the root node. This is the 'Off path with data' scenario. This method only works well if the Global tree has enough data, because otherwise...

If the player places their piece in any slot that isn't 5 or 7 as their second move, then the example Global tree doesn't have any nodes representing this meaning the AI can't use WalkPaths with StatisticallyBest to figure out the best path to take. In this case, the AI should fall back to performing completely random moves during a match. This allows it to continue playing, while still gathering data for its Global tree. This is the 'Off path without data' scenario.

Problem 3.3 – What algorithms does the AI use to handle these problems?

First, this is the algorithm 'DoRandom', which the AI uses to perform a random move.

Steps:

1. Generate a number between 0 and 8 (inclusive). This number is referred to as *index*.
2. If the slot at *index* is not empty, go to Step 1.
3. Otherwise, place the AI's piece at this slot.

Now, for the algorithm that the AI uses to determine its move. Steps that are encased in square brackets ('[' and ']') are comments.

This algorithm handles the 3 previously described scenarios. I will refer to this algorithm as the "Find Move" algorithm

Variables:

- **Parent** = The node that will be given as the root node for the WalkPaths (with StatisticallyBest) algorithm.
- **LastNode** = A single node, used when trying to mirror the local tree with the global tree.

Steps:

1. [If the Local tree has nodes in it, then 'mirror' the local tree with the Global tree, so the AI gets the Global tree's version of the nodes] If the local tree has at least 1 node in it:
 - a. Set **LastNode** to the Global tree's root node.
 - b. Get the next node from the Local tree (going from the start).
 - c. Compare the hash of the node from the Local tree with the hashes of the **LastNode's** children.
 - i. If a matching hash is found, set **LastNode** to the matching node in the Global tree, and go to Step 1.b
 - ii. Otherwise, fall back to using the DoRandom algorithm.
(This is the 'Off path without data' scenario)
 - d. Set **Parent** to **LastNode**
2. If the Local tree doesn't have any children in it, set **Parent** to the Global tree's root.
3. Perform WalkPaths, where Action is StatisticallyBest, and Root is **Parent**.
4. Get the first node from the path that the StatisticallyBest algorithm chose, and perform the move that the node represents. [This is both of the other scenarios, since the algorithm technically handles both.] *

* Because this algorithm expects the direct children of the root node to only represent the AI's moves, the first node in the path **should** represent the AI's move. If it doesn't, then something has gone wrong.

There is a flaw with this algorithm though; it expects that the first children of the root node **only** represent the AI's moves. If they represent the player's or both the AI's and the player's, the algorithm will fail.

This means the *root* must either represent no move at all (the true root node of a tree), or the move of the player since a node representing the player can only have children representing the AI, and vice-versa. This means that the AI must always make the first move, because if the AI always goes first, then the true root node of a tree will only have children representing the player, keeping the tree in the format that this algorithm expects.

Proposed structure of the program

My reasons for proposing a synchronous (multi-threaded) approach

Because the game will be making use of a window (referred to as 'GUI' for the rest of the document), the game will require the use of either asynchronous/synchronous technology. I have chosen to take a synchronous (multi-threaded) approach, as I am more familiar with it than asynchronous code.

There will be 2 main threads in the program; the 'Game thread', which runs the actual logic for the game (AI taking its move, checking if someone has won, etc.), and the other thread is the 'GUI thread' which is the thread the GUI will run on.

The reason for this approach is, imagine the AI gets to the point it has to spend several seconds to calculate a move (due to too much data to go through quickly), if the AI was running on the same thread as the GUI, then the GUI would be shown as 'Not responding' – a scenario I would like to avoid, as it makes the game feel slow and unresponsive (and seeing a game produce the "XXX.exe is not responding" message isn't a terribly great sign of a well-made program).

The solution is to run the game logic in a separate thread (The 'Game Thread') from the GUI (The 'GUI thread'), so even if the AI is taking up to a minute to calculate a move (a scenario my algorithm can't avoid, but would require **many** matches to have been played), the GUI still remains responsive, and a simple message such as "The AI is thinking" can be displayed to the user so it's more obvious what the game is doing.

Communication between threads

One of the most annoying/biggest issues with multi-threaded software is communication between threads.

C# (or rather, the .Net framework) provides a container called 'ConcurrentQueue' ^[1] which is a thread-safe (safe to use between multiple threads at the same time) implementation of a queue.

My solution for the GUI thread talking to the Game thread ('Game thread to GUI thread' will be covered later) is to provide a ConcurrentQueue that they can both access. This ConcurrentQueue should store messages, so a basic communication may look like:

GUI Thread: Queue Message 'Start Match'

Game Thread: Get a Message from the queue.

Game Thread: Sees that it is the 'Start Match' message, so executes the code to start a match.

More examples of how this messaging will work will be explained later.

Now, we have a basic solution for how the GUI thread can talk to the Game thread, so how does the Game thread talk to the GUI thread? The windows in a WPF program provide something called a Dispatcher ^{[2][3]} which to put it simply, allows threads to queue up 'tasks' to be ran in the thread that the Window itself is ran in. (slightly ironically, I'm pretty sure it uses asynchronous technology).

So, with the dispatcher, a basic communication may look like:

Game Thread: Queue up task 'Update On Screen Grid', passing data on what the grid looks like

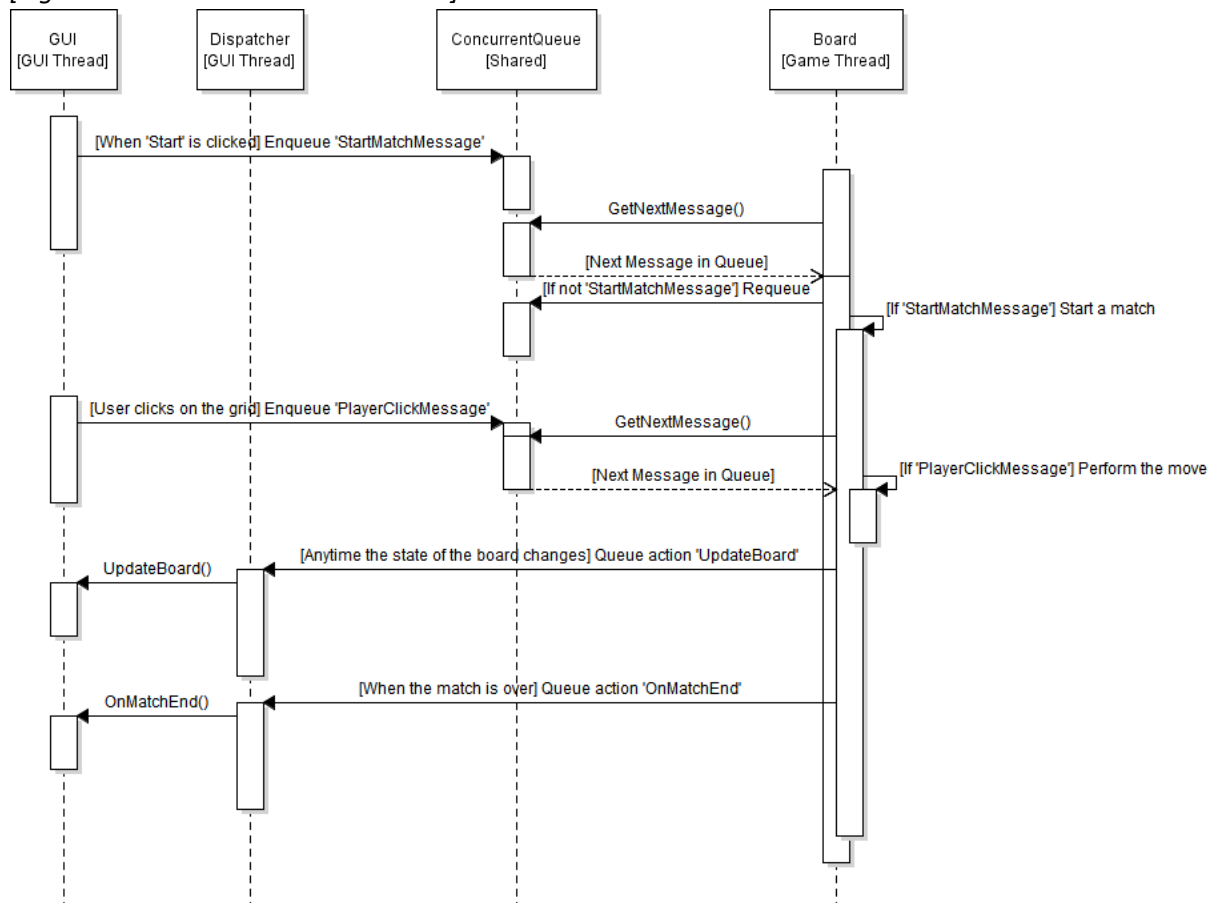
GUI Thread: [At some point] Execute task 'Update On Screen Grid'

The reason that the Game Thread can't directly execute these tasks on the GUI, is because WPF does not allow anything displayed in a window to be modified from any thread other than the thread the window is being ran on. This means that the Dispatcher is used as a 'proxy' for the Game thread to update the GUI.

Also, in WPF, I have not been able to find a reliable way to run a piece of code every 'tick' (update of the window), so this makes it difficult/slow to make the GUI thread also use the ConcurrentQueue mentioned earlier (otherwise, I'd just make both threads share the queue, and if either thread got a message they shouldn't have, they just requeue it. Or at worst, supply one queue for 'gui->game', and one queue for 'game->gui').

Here is a diagram depicting how the two threads may communicate:

[Figure: Cross-thread communication]



[1] [https://msdn.microsoft.com/en-us/library/dd267265\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/dd267265(v=vs.110).aspx)

[2] [https://msdn.microsoft.com/en-us/library/system.windows.threading.dispatcher\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.windows.threading.dispatcher(v=vs.110).aspx)

[3] [https://msdn.microsoft.com/en-us/library/system.windows.window\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.windows.window(v=vs.110).aspx)

The Game loop

My idea of how the game logic should work, is that there is a 'Board', and that there are 'Controllers'.

The 'Board', as the name suggest, provides an interface to modify a tic-tac-toe game board. The 'Board' will also contain the logic for executing a match.

The 'Controllers' can be viewed as the 'Players'. There is a controller for the 'X' piece, and a controller for the 'O' piece. There will be a controller that handles the input of the player (explained later), and a controller which provides the AI.

My reasoning for having 'Controllers', is that it allows my code to be more reusable and modular. For example, I could create a match between two player controllers to represent a 'Player versus player' match; I could put up two AI controllers against each other to have the AI fight itself, and of course, I could put a player controller up against an AI controller. It also lends way to further additions, as an example if I ever wanted multiplayer in the game, I could just create an 'OnlinePlayer' controller that handles getting input from over a connection and put it up against another player controller.

The use of a hash to represent the board state (explained earlier in the document) will allow controllers to be written in a 'Piece-independent' way, meaning the code can be written in a way that doesn't matter whether the controller is 'X', or 'O'. Although, as explained previously alongside the AI's algorithm, the AI controller will always have to be the piece that gets the first move (this is an issue with the algorithm the AI uses, not an issue with the hash/program structure/whatever else).

Controllers should be given information about the state of the match at the following steps, where each step is named with text inside of square brackets:

- [OnStart] When a match is first started, the piece that the controller is assigned to should be passed to it. This gives controllers a chance to setup whatever they need.
- [OnDoMove] When the controller needs to perform its move, a hash of the board (hashed from the point of view of the controller), as well as the index of where the last piece was placed should be passed. Incidentally, the index passed represents the index of where the enemy controller placed its piece since the controllers take it in turns to place a piece.
- [OnAfterMove] After the controller has performed its move, the hash of the board and the index of where the controller placed its piece should be passed. This allows controllers to separate their logic to update their internal state from the logic to perform a move.
- [OnEnd] When the match ends, the hash of the board, the index of the last piece placed, and the result of the match (win, loss, tie) should be passed to the controller. This gives controllers a chance to, for example, update the GUI to say "You have won", or for controllers such as the AI controller to save data for future matches.

The Board will contain the logic of performing a match, and the algorithm for performing a match is defined as:

Variables:

- **XCon** = The controller that represents the X piece.
- **OCon** = The controller that represents the O piece.
- **TurnPiece** = If 'X', then it's **XCon's** turn. If 'O', then it's **OCon's** turn.
- **LastIndex** = The index of where the last piece was placed on the board.

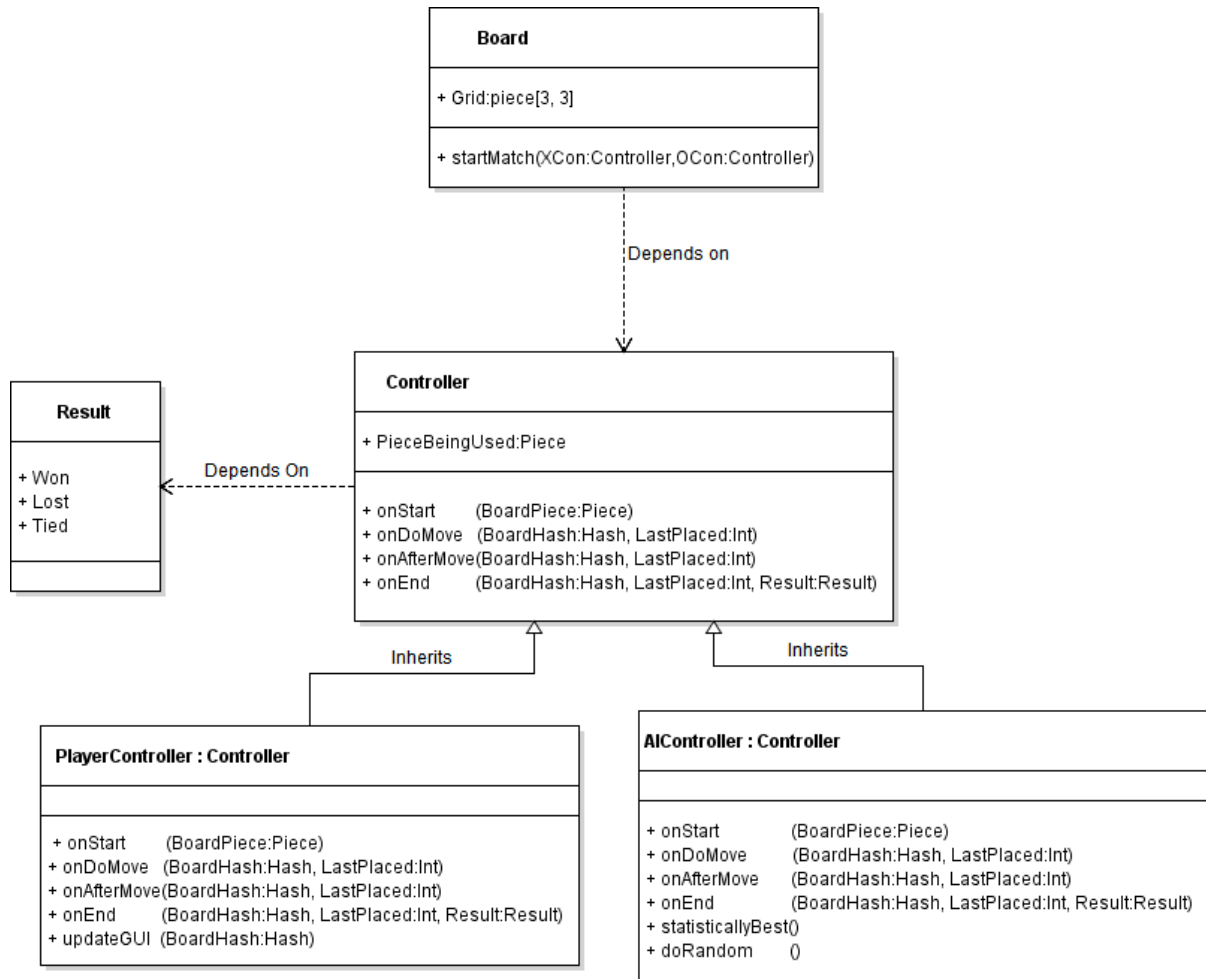
Steps:

1. Perform the OnStart step for both controllers, passing 'X' to the **XCon**, and 'O' to the **OCon**.
2. Default **TurnPiece** to either 'X' or 'O'.
3. Default **LastIndex** to some value, this value represents 'No pieces have been placed yet'.
4. Perform the OnDoMove step for the controller whose turn it currently is (based on **TurnPiece**). A hash of the current state of the board (where 'M' represents the controller's piece, and 'O' represents the enemy controller's piece) is given, as well as the index of where the last piece was placed.
5. Perform the OnAfterMove step for the controller whose turn it currently is, where a new hash of the board is given, and the index of where the controller placed its piece is given (so it doesn't have to keep track of it itself. In code, this will create less bugs).
6. Check to see if someone has won, or if there is a tie. If no one has won yet, and there isn't a tie, then:
 - a. Set **TurnPiece** to 'X' if its currently 'O', or set it to 'O' if its currently 'X'.
 - b. Go to Step 4.
7. Perform the OnEnd step for both controllers, where a new hash of the board is made for both controllers (from their own points of view), and passing whether they won, lost, or tied.

One final benefit of going with the concept of a controller, is that the Board **only** focuses on the 'rules' of the game, whereas the controllers are what provide the AI/interaction for the player, creating a separation of responsibilities, leading for less places for bugs to pop up (e.g. if the AI has something wrong with it, then there is a very high chance that the issue is with the AI's controller, not the Board, since the Board doesn't even really do anything specifically for the AI).

Here is a diagram to further demonstrate the relationship between the Board and its Controllers, with two example controllers given to show the modularity of this setup.

[Figure: Relationship between the Board and the Controllers]



The Game Thread

All that's really left to talk about the Game thread itself (including the Board, but not any specific controller) is how it starts a match.

It's simple enough to be explained in an algorithm:

Variables:

- **Queue** = The ConcurrentQueue shared between the GUI and Game thread.

Steps:

1. Get a message from the **Queue**.
2. If this message is 'Start Match', then start a match between the two controllers that the message provides. (This starts the algorithm for the Board's game loop).
 - a. Once the match is over go to Step 1.
3. If this message isn't any of the above, requeue it into the **Queue**.
4. Sleep the thread for 50 milliseconds.
5. Go to Step 1.

In the actual project, I may need to create more messages for the above algorithm to handle, but for a simple 'Start Match' message it works well.

As a note, step 4 makes the thread sleep for a small amount of time, because when turned into code the algorithm is a 'while(true)' loop and can eat up a lot of CPU time just constantly checking for a message, so a sleep is inserted to make it only check once every 50ms, instead of checking non-stop.

The final thing to talk about it is how the game thread gets terminated. In C# (or .Net, to be precise) there is an option to 'abort' a thread ^[1] which will cause a special exception to be thrown in the thread. I plan on using this to make the GUI abort the Game thread when the GUI is closed. If the Game thread didn't close, then the game's process would still run in the background, even after the GUI is closed. I admit I could design this better, so there is some "doGameThread" flag, but there isn't much to gain from it in this case.

[1] [https://msdn.microsoft.com/en-us/library/5b50fdsz\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/5b50fdsz(v=vs.110).aspx)

The GUI Thread

The GUI itself should at the very least provide a 3x3 grid (representing a tic-tac-toe board), which contains an up-to-date view of a match, and should allow the player to interact with it. More information about what the GUI displays is described in the ‘Success Criteria’ at the start of the document.

WPF is an event-based framework, meaning doing things such as “Click button”, “Move mouse over text”, etc. can fire an event in the code. So, the 3x3 grid in the GUI should be made up of buttons (or just plain text) where their “On click” event sends a message through the ConcurrentQueue.

For example, if the user presses the top-right slot of the grid, then a message that says ‘Player Piece at slot 2’ should be sent. This is then handled by the player’s controller on the game thread, which will read in the message, and if the move is valid (it’s the player’s turn, and the slot is empty) then it is performed, but if the move is not valid then it simply does nothing.

The GUI itself should provide an interface for the Game thread, which allows the Game thread to modify the GUI (update the grid, change some text on screen, signal whether a game is playing or not, etc.) via the use of the GUI’s dispatcher.

The GUI is also responsible for setting up the Game thread, because in a WPF program, the main window’s (the GUI) constructor is generally the entry point of the program.

The GUI should provide a ‘Start Match’ button, which has the “On Click” event of sending the ‘Start Match’ message described earlier. The two controllers for this message are the AI controller, and the other being the Player controller.

The AI Controller

The AI Controller is the controller that will provide the logic for the AI.

[OnStart]

When a match is started, the AI should make sure it has loaded the Global tree from a file (if it hasn’t done it already).

[OnDoMove]

When it is the AI’s turn to perform a move, it needs to do two things.

1. The *Hash* and *index* given to it can be used to figure out which move the enemy controller last made. This move should be added into the local tree.
2. The AI should then perform its ‘FindMove’ algorithm, to perform its move.

[OnAfterMove]

When the AI has completed its move, it should add its move to the local tree.

[OnEnd]

When the match has ended, the AI should bump either the ‘win’ or ‘loss’ counter of each node in the local tree, then merge the local tree into the global tree.

The AI may also save its Global tree into a file at this stage.

The Player Controller

The Player Controller is the controller that allows the user to interact with the game board, as well as being responsible for displaying the current state of the match to the user.

During a match, the player controller will make use of the GUI's dispatcher (explained earlier), as well as an interface provided by the GUI so that the controller can update it.

[OnStart]

When a match is started, the controller should pass which piece the player is playing as to the GUI, so the GUI can then update itself to display something such as "You are playing as X".

[OnDoMove]

During the player's turn, it should first pass the current *Hash* of the board to the GUI, so it can display to the user the current state of the board. It should also tell the GUI that it's the player's turn, so it can display "It is your turn" on screen, as well as allow the user to interact with the board.

The controller should then keep checking for a 'PlayerPlaceMessage' in the ConcurrentQueue (also explained earlier). This message will contain the index of where the player wants to place their piece. If the move is invalid (the slot is non-empty), then the controller will ignore it, and wait for another message. Otherwise, it will perform the move.

[OnAfterMove]

After making its move, the controller should send the new *Hash* of the board to the GUI, to keep it up to date. It should then tell the GUI that it is the AI's turn, so it can stop the player from creating 'PlayerPlaceMessages', as well as display "It is the AI's turn".

If the player could create messages when it wasn't their turn, then the messages would 'buffer', allowing the user to queue up any number of moves to automatically be performed. The issue is, most users won't find a use for this, and instead if they click one too many times they'll find that their next turn is basically 'skipped'.

[OnEnd]

As with the other stages, the AI should send the final *Hash* of the game board to the GUI, and then also inform the GUI that the match is over, alongside with the match's *Result* so it can display "You have won/lost/tied" as appropriate.

Usability

The usability features of the game involve how easy/accessible the game's GUI is to use.

The GUI itself will contain a 3x3 grid, which mimics the kind of 3x3 grid that would be used in a real-life game of tic-tac-toe. This means that anyone familiar with the game will be able to easily understand how to interact with the GUI.

The only action that needs to be performed on the GUI, is clicking. The user will have to click a 'Start Match' button to begin a match, they also have to click on the 3x3 grid to place their piece on the screen. This means that there is virtually no barrier of entry to play the game, and even a child who is too young to read would be able to play it, in contrast to if I went with a command-line interface for the game, where the user would have to type in commands to play.

The GUI should display which piece the user is playing as (to avoid confusion), as well as whether it's the player's turn, or the AI's turn. This is mainly to prevent confusion to the player.

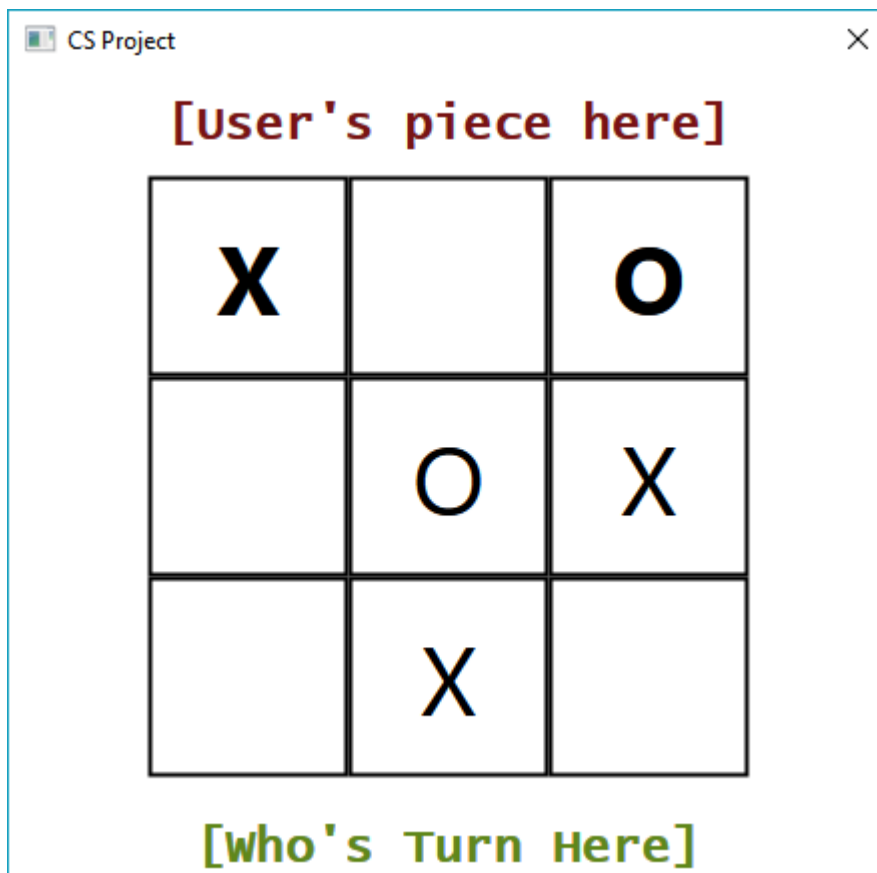
Because the GUI and the game logic run on separate threads, in the event that the game logic is taking a while to process something (such as the AI's logic), then the GUI will still be fully responsive to the user. This prevents the GUI from freezing, which would make it feel slow, unresponsive, and buggy.

Each cell in the 3x3 grid should be large enough to see which piece is in it, and to be easy to click. The font used to display the pieces in a grid should be clean and simple, where the 'X' and 'O' characters are easy to read. The font may also be **bold** to make it easier to see the pieces.

The image below is a small mock-up GUI I have put together. It uses WPF, so this mock-up is representative of what the final GUI could look like (it may even become the final GUI itself). Some things to note are:

- The font used for the 3x3 grid is Segoe UI, as it is a very clean and clear font.
- Each cell in the grid is 100x100 pixels. There is no particular reason for this, except that I found this size to be the nicest to look at. Other sizes tried were 50x50, 125x125, 150x150, and 200x200.
- The top row of the grid uses emboldened text, while the rest use non-bolded text (to see which is easier to look at).
- The "[User's piece here]" and "[Who's turn here]" text are coloured differently so they stand out.
- The font used for the two pieces of text mentioned above is 'Lucida Console'. It is different from the 3x3 grid so it stands out.
- The window itself is 455x460, so should be able to display properly on most monitors.

[Figure: Mock-up GUI]



To aid users who may not know about what tic-tac-toe is, the game should display a message box detailing how to play tic-tac-toe when it is loaded for the first time. There should also be a 'Help' button somewhere so the user can bring the message box back up.

The cells of the grid should change colour whenever the player hovers their mouse over them (and change back to their original colour afterwards). This creates a sense of interactivity to the user.

Touch screen monitors should work fine with the game, as the cells are big enough to comfortably tap, and since touch screen monitors basically turn your finger into a mouse, the same code can be used for both cases (meaning less places in the code for bugs).

Test Data for development

Test Data for Development will be a set of automated unit tests that test specific parts of the code to ensure they are working correctly.

Some tests I may create during the development of my project include:

Action	Expected Result
<p>Perform a match between two controllers, with a predetermined set of moves.</p> <p>'M' is for Controller #1, 'O' is for Controller #2. This is the order of moves to happen over the match:</p> <p>"[. . .] [. . .] [. . .]" -> "[M . .] [. . .] [. . .]" -> "[M . .] [O . .] [. . .]" -> "[M M .] [O . .] [. . .]" -> "[M M .] [O O .] [. . .]" -> "[M M M] [O O .] [. . .]"</p>	<p>Controller #1 is passed 'Result.Won' to its 'onEnd' function, while Controller #2 is passed 'Result.Lost'. This tests that the Board's win check logic is working (for winning and losing).</p> <p>That, for both controllers, the hash and index passed to 'onDoTurn', 'onEnd', and 'onAfterTurn' are the correct values. This checks that the Board is keeping track of the right values, and passing them to the controllers properly.</p> <p>For example, on move #3 (the list of moves starts from 1, to 2, to 3, etc.) Controller #2 for its 'onDoTurn' function should be passed the hash "M....." (reflecting move #2) and an index of 0 (where the Controller #1 placed its piece).</p> <p>On the side, this also tests to make sure the Board will correctly place pieces into the grid (at least, for the ones being used).</p>
<p>Perform a match between two controllers, with a predetermined set of moves.</p> <p>An example hash of the final state of the board, which results in a tie:</p> <p>"[M O M] [O M O] [O M O]"</p>	<p>The result of the match should be a tie between the two controllers. 'Result.Tied' should be passed to the 'onEnd' functions of the controllers. This tests the tying logic for the win check code.</p>

Action	Expected Result
<p>Serialise a pre-made tree of nodes to a file, and then load it back in.</p> <p>The reason a pre-made tree should be used is so testing is easier/more consistent than using a randomly made tree. Although, a randomly made tree does have a more likely chance of discovering bugs in the code (since there would be more data in the tree).</p>	<p>That when loaded back in, the data of the loaded tree is exactly the same as the data of the tree that was saved.</p> <p>This tests the code to save/load a tree of nodes to/from a file.</p> <p>An example would be to serialise this tree: -> "OM....."[W:1 L:3 I:1] "O....."[W:2 L:4 I:0] -> "O.M....."[W:1 L:1 I:2]</p> <p>After serialising the tree, load it back in, and check to see that all of the data (the hashes, the win/loss counter, and the index) are all the same as before.</p>
<p>Using a pre-made tree, test the implementation of the WalkPaths algorithm, where <i>Action</i> is the implantation of StatisticallyBest.</p>	<p>As an example, imagine the pre-made tree contained 3 paths: A path with a 20% win rate A path with a 40% win rate A path with a 60% win rate</p> <p>The expected result is that the third path, with a 60% win rate, is selected by the algorithms.</p> <p>This first of all tests that the WalkPaths implementation is working correctly. This then also tests that the StatisticallyBest implementation is working.</p>

Test Data for beta testing

Test Data for beta testing will be a set of tests that must be performed manually, usually because writing a unit test for it would be impractical.

Some tests I may create include:

Action	Expected Result
The user clicks on the on-screen grid while no match is being performed.	Nothing happens. This is more to check that the GUI/player controller isn't misbehaving in some way, rather than testing specific functionality.
The user opens the game, and then closes the game.	No random crash/exception message should pop up on screen. Again, this is to check for malfunctioning code, rather than testing specific features.
The user presses an on-screen button that says 'Start Match'.	<p>The GUI should send a message to the Game thread to start up a match between the AI and the player.</p> <p>This is to test that the game thread is correctly listening for messages, that the GUI sends the 'StartMatch' message, and the code for performing a match starts up properly.</p>
The user presses a slot on the on-screen grid, where the slot already has a piece in it.	<p>Nothing. The game should simply do nothing in this case, and allow the player to continue trying to place their piece.</p> <p>Again, this is to test for misbehaving code.</p>
The user spams their left mouse button on the GUI.	Nothing. This is just to see if clicking like a maniac somehow uncovers some obscure bug.
The user presses the 'Start Match' button, and that the game displays which piece the player is using, and whose turn it is.	<p>The GUI is displaying the correct piece that the player is using (if the player plays as 'X', then it shows 'You are X', for example).</p> <p>The GUI should also be correctly displaying whose turn it is.</p>
The user opens the game for the first time, closes the game, then opens it again.	<p>The first time the game is opened, the game should notice that a 'ShowMessage' flag is not set, so displays a message box detailing what tic-tac-toe is and how to play it. It should then set this 'ShowMessage' flag.</p> <p>The second time the game is opened, the game should notice that the 'ShowMessage' flag is set, so no longer displays the message box automatically.</p>

Development

Iterations of development, prototypes, and testing

During the development of my project, I kept a separate document which I called “Roadmap”. This roadmap has been added to the end of this report, and is included in the table of contents. I mention the fact that it was a separate document, as it’s formatting and overall style might seem noticeably different than the main report’s.

The roadmap is where I detailed the goals for specific versions of the project, and it also keeps a list of all the minor versions leading up to the major ones, detailing what each version added alongside my thoughts and comments for each version.

The roadmap ends with a chronological list of changes made with the project (that were worth noting). This should be sufficient evidence of iterative development, and that prototype versions of the project existed.

Each version in the roadmap also comes with a screenshot of the results from its unit tests, providing evidence of testing.

Evidence of modular code

Just as a note, the complete code (including testing code) for the solution has also been added as an appendix to the end of the document.

I would first like to reference back to the “Board + Controllers” concept as the prime example of modular code in the project (e.g. putting two player controllers against each other, two AI controllers against each other, and a player controller against an AI controller).

This is achieved via the ‘Controller’ base class, which the ‘PlayerGUIController’ (controller that interacts with the GUI) and the ‘AI’ (controller that implements the AI) both inherit from, allowing them to be used with the ‘Board’ class’ ‘Board.startMatch’ function in a generic and fully modular way.

[The signature for the ‘Board.startMatch’ function]

```
/// <summary>
/// Starts a match between two controllers.
///
/// Note to self: Run all of this stuff in a seperate thread, otherwise the
GUI will freeze.
/// Use System.Collections.Concurrent.ConcurrentQueue to talk between the
two threads.
/// </summary>
///
/// <param name="xCon">The controller for the X piece.</param>
/// <param name="oCon">The controller for the O piece.</param>
public void startMatch(Controller xCon, Controller oCon)
```

[The truncated code in 'MainWindow' that generates the 'StartMatch' message. In this case it puts a PlayerController up against the AI controller]

```

/// <summary>
/// This function is called when the 'Start Match' button is pressed.
/// It begins a match between the AI and the player.
/// </summary>
private void onStartMatch(object sender, RoutedEventArgs e)
{
    [...truncated]
    // Then, start up a match between the AI and the player
    if(this._aiInstance == null) // Use only a single instance of the AI,
    so it doesn't have to reload the global tree over and over.
    {
        try
        {
            [...debug-only code removed]
            this._aiInstance = new AI(null, null);
        }
        catch(Exception ex)
        {
            [...]
            return;
        }
    }

    // Then send a message to start a match.
    this.gameQueue.Enqueue(new StartMatchMessage
    {
        xCon = new PlayerGUIController(this),
        oCon = this._aiInstance
    });
}

```

[A snippet of the game thread's 'main' function, where it looks for the 'StartMatch' message]

```

// Check for a message every 0.05 seconds.
Message msg;
while (!this.gameQueue.TryDequeue(out msg))
    Thread.Sleep(50);

// If we get a StartMatchMessage, then perform a match.
if(msg is StartMatchMessage)
{
    var info = msg as StartMatchMessage;

    state = GameState.DoingMatch;
    board.startMatch(info.xCon, info.oCon);
    state = GameState.Waiting;
}

```

As an example of the 'Board + Controller' setup being modular, the 'Board' class itself has no concept of the GUI (despite the self-notes in some comments for its code), but the 'Controller' base class provides enough opportunities (onStart, onEnd, onDoTurn, and onAfterTurn) for the 'PlayerGUIController' (which does understand what the GUI is) to keep an up-to-date view of the game board's state on the GUI.

The use of a ConcurrentQueue coupled with Messages (there is also a base 'Message' class that all of the more specific messages inherit from) eases the way of future rewrites/structural changes.

For example, the 'PlayerGUIController' waits for a 'PlayerPlaceMessage' from the queue, and then uses the information in that message to perform its next move. However, it doesn't actually care about where this message comes from, so in the future if say, I wanted to make a command line interface for the game (as opposed to using WPF), then all I would have to do is make the new command line front-end generate the 'PlayerPlaceMessage', and, without any changes to the class itself, the 'PlayerGUIController' would work like normal.

The example used above could also make it possible to create automated tests for the 'PlayerGUIController' (by fabricating 'PlayerPlaceMessages') but I ultimately decided to not do that (it's obvious enough just by playing the game if the controller isn't working in some way).

[The code in 'MainWindow' that generates a 'PlayerPlaceMessage'. As a note, the name of the labels(slots on the grid) follows the pattern 'slot0', 'slot1', etc. up to 'slot8']

```
// When one of the slots are pressed, send a PlayerPlaceMessage to the game
// thread, saying
// which slot was pressed.
private void onSlotPress(object sender, MouseEventArgs e)
{
    // Don't do anything if the player isn't allowed to place a piece
    yet.
    if ((this._flags & Flags.CanPlacePiece) == 0)
        return;

    // Lock the game board
    this._flags &= ~Flags.CanPlacePiece;

    // Only labels should be using this
    var label = sender as Label;

    // The last character of the labels is an index.
    var index = int.Parse(label.Name.Last().ToString());

    this.gameQueue.Enqueue(new PlayerPlaceMessage { index = index });
}
```

[Finally, the snippet of code for the PlayerController looking out for the message.]

```
// Wait for the GUI to have signaled that the player has made a move.
Message msg;
while(true)
{
    // Check every 50ms for a message.
    // If we didn't use a sleep, then the CPU usage skyrockets.
    if(!this._window.gameQueue.TryDequeue(out msg))
    {
        Thread.Sleep(50);
        continue;
    }

    // If we get a message not meant for us, requeue it.
    if(!(msg is PlayerPlaceMessage))
    {
        this._window.gameQueue.Enqueue(msg);
        continue;
    }

    // Otherwise, see if the placement is valid, and perform it.
    var info = msg as PlayerPlaceMessage;
    if(!boardState.isEmpty(info.index)) // Make sure the slot is empty
    {
        this._window.unlockBoard(); // Unlock the board, otherwise the
game soft-locks
        continue;
    }

    // Place our piece at the index in the message
    this.board.set(info.index, this);
    break;
}
```

Evidence of validation

In the code appendix, I have attached the testing code used for the automated unit tests used throughout the project.

Because the testing code is poorly commented (a flaw with my project, I realise), I will explain each test in the program. These tests, while they cover very important parts of the program, fail to cover enough different use cases for the code they test, so some bugs that should've been very easy to catch with tests ended up slipping into the project.

[Figure: List of unit tests]

Passed Tests (10)	
✓ basicMatchTest	19 ms
✓ hashSerialiseTest	671 ms
✓ mergeTest	4 ms
✓ nodeSerialiseTest	44 ms
✓ removeTreeTest	41 ms
✓ saveTreeLoadTreeTest	155 ms
✓ statisticallyBestTest	5 ms
✓ testHash	132 ms
✓ treeExistsTest	692 ms
✓ walkTest	2 ms

Test Name	Description
basicMatchTest	<p><i>[This tests the basic functionality of the 'Board' class]</i> <i>[Code appendix file: Unittests/BoardTests.cs]</i></p> <p>Performs a dummy match between a "NullController" and a "StupidController".</p> <p>The NullController places its pieces in the middle row of the grid (left to right), while the StupidController places its pieces in the top row of the grid.</p> <p>The controllers checks that the data being passed to them is what's expected, and it also tests the win condition logic of the 'Board' class (though, I should've tested every single win position, to make sure they're coded in properly).</p>
hashSerialiseTest	<p><i>[This tests the serialisation function of the 'Hash' class]</i> <i>[Code appendix file: Unittests/GameHash.cs]</i></p> <p>This test serialises two different hashes (one that uses 'O' as the 'M' piece, and one that uses 'X' as the 'M' piece, to ensure both pieces of data are encoded correctly) to a file, and then loads them back from the file, checking all of the information has been retained.</p>

Test Name	Description
mergeTest	<p><i>[This tests the function used to merge two trees of nodes together]</i> <i>[Code appendix file: Unittests/NodeTests.cs]</i></p> <p>This test creates two handmade tree (Tree #1, and Tree #2) where Tree #2 is merged twice into Tree #1. The merge happens twice to test that the merge function successfully handles nodes that don't exist (first merge), and nodes that already exist (second merge). The first tree's data is then checked to ensure that the data is as expected.</p>
nodeSerialiseTest	<p><i>[This tests the serialisation function of the 'Node' class]</i> <i>[Code appendix file: Unittests/NodeTests.cs]</i></p> <p>This test is similar to the hashSerialiseTest, where a tree is serialised, then deserialised to/from a file, and then its data is checked to make sure everything is correct.</p>
removeTreeTest	<p><i>[This tests the function used to remove an already existing .tree file]</i> <i>[Code appendix file: Unittests/GameFilesTests.cs]</i></p> <p>This test first saves a dummy tree to a file, checks that it exists, removes the dummy file, then checks that it doesn't exist. The game in its current state does not make use of the function this test is testing, but at least I know it works for when I want it.</p> <p>This test also makes sure that the function being tested throws an exception if the tree to be removed doesn't actually exist.</p>
saveTreeLoadTree	<p><i>[This tests the functions used to save and load trees to files]</i> <i>[Code appendix file: Unittests/GameFilesTests.cs]</i></p> <p>This test is similar to the nodeSerialiseTest, where a tree is saved to a file, then loaded back in, having its data checked.</p> <p>This test makes sure the save function throws an exception if the tree already exists, and the overwrite flag is false, and if a null tree is passed to be saved.</p> <p>The load function is checked to make sure it throws an exception if a non-existent tree tries to be loaded.</p>
statisticallyBestTest	<p><i>[This tests the function that implements the 'StatisticallyBest' algorithm]</i> <i>[Code appendix file: Unittests/AverageTests.cs]</i></p> <p>This test creates a tree with 3 paths, one path has a 50% win chance, one has a 62.5% win chance, and the last has a 75% win chance. This function tests to make sure that the statisticallyBest algorithm selects the 75% win chance path.</p>
testHash	<p><i>[This tests most of the basic functions of the 'Hash' class]</i> <i>[Code appendix file: Unittests/GameHash.cs]</i></p> <p>This test tests the basic functionality of the Hash class, and tests for all of the different exceptions that these basic functions can throw.</p>
treeExistsTest	<p><i>[This test is functionally identical to the removeTreeTest]</i></p>

Test Name	Description
walkTest	<p><i>[This tests a function that implements an algorithm not explained in this document. It is basically a generic version of the algorithm the AI uses to merge the local tree into the global tree]</i></p> <p><i>[Code appendix file: Unittests/NodeTests.cs]</i></p> <p>The function being tested, when given a list of hashes, will attempt to walk a path where each node's hash matches the hashes of the given list. The test just makes sure that the path it creates is correct.</p>

The tests, despite covering less possible paths in the code than I'd like, have generally always been passing since they were coded. The only times I can recall of the tests failing, is when I messed up some part of the testing code itself, or if I was changing some code and running the test(s) for it to see if it still passes.

A lot of the code in the project also has error checking code to ensure that the given parameters are fine to use. One example is the function 'Board.set' which has 5 separate checks to ensure that it is being used properly:

```
Public class Board[Board.cs in the appendix]:

/// <summary>
/// Sets a piece on the board.
/// </summary>
///
/// <param name="index">The index of where to place the piece.</param>
/// <param name="controller">The controller that's placing the
piece.</param>
public void set(int index, Controller controller)
{
    // There are so many ways to use this function wrong...
    // But I need these checks here to make sure my code is correct.
    Debug.Assert(this._stage == Stage.InControllerTurn,
        "A controller attempted to place its piece outside of its onDoTurn
function.");

    Debug.Assert(this.isCurrentController(controller),
        "Something's gone wrong somewhere. An incorrect controller is being
used.");

    Debug.Assert(index < Board.pieceCount && index >= 0,
        $"Please use Board.pieceCount to properly limit the index. Index =
{index}");

    Debug.Assert((this._flags & Flags.HasSetPiece) == 0,
        "A controller has attempted to place its piece twice. This is a
bug.");

    Debug.Assert(this._board[index] == Piece.Empty,
        "A controller attempted to place its piece over another piece. Enough
information is passed to prevent this.");

    this._board[index] = controller.piece;
    this._lastIndex    = index;
    this._flags        |= Flags.HasSetPiece;
}
```

Another example, which is more common throughout other pieces of the code, is where the function does a simple check to make sure that it isn't passed something such as a null value. This is a slightly truncated snippet of the 'GameFiles.saveTree' function that demonstrates this check.

```
Public class GameFiles[GameFiles.cs in the appendix]:

/// ...
public static void saveTree(string name, Node root, bool overwrite = true)
{
    if(root == null)
        throw new ArgumentNullException("root");

    if(!overwrite && GameFiles.treeExists(name))
        throw new IOException($"Unable to save tree {name} as it
already exists, and overwrite is set to false.");

    // Make sure the tree directory exists, then serialise `root` into a
    new tree file.
    [irrelevant code here]...
}
```

One final example is a private function in the 'Hash' class, which is called by most of the other functions in the class:

```
Public class Board.Hash[Board.cs in the appendix]:

/// <summary>
[Non-important text here...]
/// This allows me to be confident that the class is working as it should
be.
/// </summary>
private void checkCorrectness()
{
    Debug.Assert(this._hash.Length == Board.pieceCount,
        $"The length of the hash is {this._hash.Length}
when it should be {Board.pieceCount}");

    Debug.Assert(this._hash.All(c => (c == Hash.emptyChar || c ==
Hash.myChar || c == Hash.otherChar)),
        $"The hash contains an invalid character. Hash =
{this._hash}");
}
```

A final note about testing, is that I almost regretted not properly following my test plan. I decided to change the logic involved with the code involved with letting the user click on the on-screen grid to place their move, which included the code being able to "lock" and "unlock" the board to the player's inputs (to solve another bug, which at the time had a very 'muddy' workaround).

When I added this change, I neglected to re-check what would happen should I click on a slot that is not empty, which uncovered a bug where the user's piece isn't placed (because it's a non-empty slot) **but** they could no longer place their piece anywhere on the board (because the code locked the board, but never unlocked it). This led to the game "soft locking", where the game was still running, except the player's turn would never end thus meaning the match itself could never end until the game is restarted.

The only reason I found out about the bug is by accident, since I miss-clicked in a non-empty slot rather than an empty one, then noticed my piece wouldn't be placed in anywhere on the board. Had I put in enough effort to make an automated test, or rather just follow the test plan properly, then this would've been caught a bit earlier than it did, and wouldn't have risked the possibility of the bug slipping past me.

Review

Each version in the roadmap comes accompanied with my comments on what the version achieved, alongside any additional notes about the version.

Evaluation

Testing and success criteria

As a reminder, this is the success criteria I defined for the project:

- A user-friendly, responsive GUI that provides: the 3x3 grid with an up-to-date view of the game board's state; text informing the player which piece they're playing as; text that displays whether it is the player's or AI's turn, and it must allow the player to place their piece via the 3x3 grid.
- An AI that is not impossible to win against, and is capable of analysing the data from its past matches to determine which move it should take.
- The game must not crash unexpectedly, and in the event that something goes wrong, it must simply show the user an error box saying something's gone wrong.

The game must be stable and free of any major bugs (for example, if the GUI suddenly stopped functioning, this is a major bug and should not happen). Certain features of the game, and small parts of the code can and should be tested. The preferred method of testing is unit testing, where a small piece of code is written to test a very specific part of the code. Features of the game that are tricky to test via code (such as how the GUI functions) should be manually tested and documented.

I gave 3 of my friends a copy of the program, a copy of the "Test Data for beta testing" table, and a copy of the success criteria.

First, the issues found by them testing the program:

1. The help box that appears says that you, the player, plays as the 'O' piece, when in reality you play as 'X'. **[Minor; GUI error]**
2. The game stores its data in a folder named 'data'. However, an oversight on one of the functions using this folder causes the program to simply not open if the 'data' folder doesn't exist. **[Major; Bug]**
3. The 'Start Match' button sometimes covers some text at the bottom of the screen. **[Minor; GUI error]**

The bug described in issue #2 could've been avoided if the test for the mentioned function was more complete. This is the function where the bug occurred:

```
Public static class GameFiles[GameFiles.cs]:

/// <summary>
/// This function determines if the game should display the help message
box.
///
/// If this function returns `true`, a file is created in the 'data' folder
which will then
/// make this function always return `false` while the file exists.
/// </summary>
/// <returns>`true` if the help box should be displayed. `false`
otherwise.</returns>
public static bool shouldShowHelpMessage()
{
    [As a note, 'GameFiles._helpFileFlag' is the string
'data/help_was_shown']

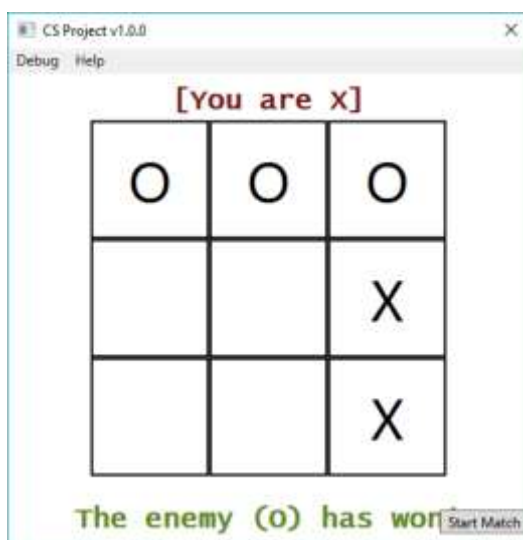
    if(!File.Exists(GameFiles._helpFileFlag))
    {
        File.Create(GameFiles._helpFileFlag).Dispose();
        return true;
    }

    return false;
}
```

The issue is, 'File.Create' will fail if the folder to create the file in doesn't already exist. The GameFiles class provides an 'ensureDirectories' helper function, to ensure that the 'data' folder (alongside another folder) exist. However, I forgot to call this function, meaning 'File.Create' would throw an exception if the 'data' folder didn't exist.

While I have manually tested in the past to see if the 'data' folder is created, because the 'shouldShowHelpMessage' function is one of the last pieces of code I wrote for the program, the bug simply slipped by undetected.

[Figure: Start Match button hiding Text on the GUI, shrunk for page formatting reasons.]



While it is unrelated to the criteria that the AI should not be unbeatable, there were two complaints I received.

The first complaint was that the AI was described as being far too easy at the start. I don't see this as a valid complaint, considering that's to be expected of an AI that learns by playing. My friends were understanding of the issue after I described this to them, so in the future I may add a note inside of the help box to explain this part of the AI.

The second complaint was that the AI, after a few matches, would always start in exactly the same position every match. This is actually another flaw with the algorithm the AI uses to select its move. During development I had picked up on this issue, and my attempt to fix it was to give the AI a 25% chance to perform a completely random move (the 'doRandom' algorithm) if the path it has selected has a 25% or less chance of winning.

This meant that the AI wouldn't lock itself out of a new path if it led to a loss the first time (which earlier attempts to solve this issue caused), but also meant that it would attempt to find a new way to beat the player if it was going down a path that would most likely make the AI lose. As a side result, this means that if the player beats the AI enough times, it will *eventually* start in a different position if the win chance becomes low enough. I admit this is not a perfect solution, but it is the one that worked the best.

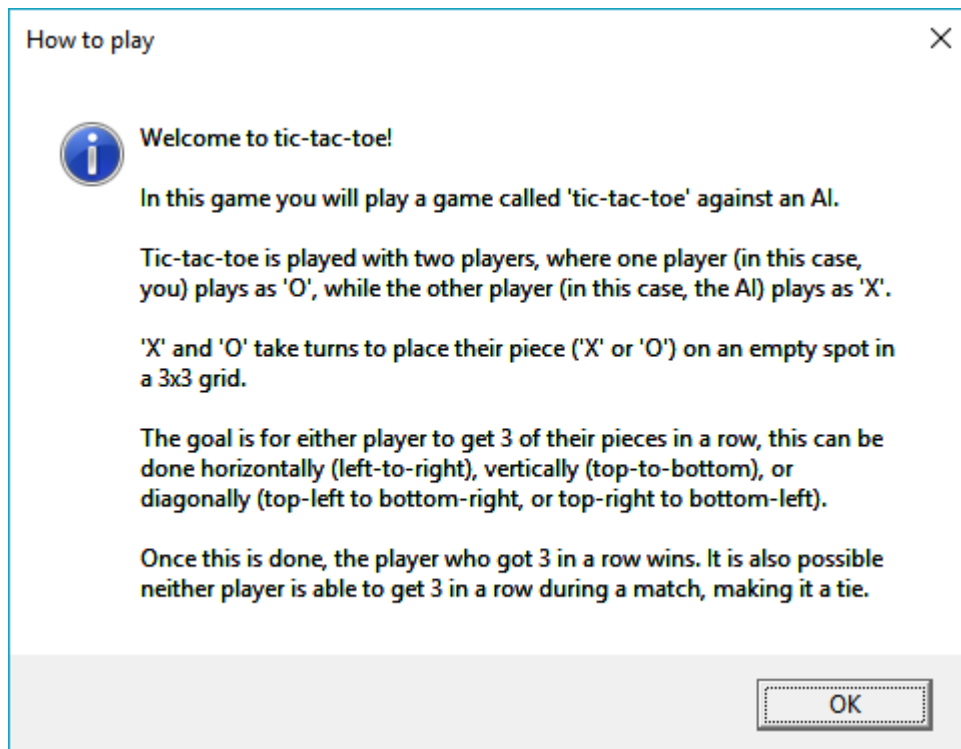
Overall, in terms of whether the project meets the success criteria:

- The GUI, while it is user-friendly, responsive, and easy to use, the aforementioned issue of the 'Start Match' button covering some of the text on the screen gives the impression that the GUI is incomplete.
- The AI is not impossible to win against (quite the opposite until it has a decent amount of data), and is able to use the data of past matches to influence its decision of what move to make.
- The game *does* contain a major bug (the 'data' folder bug), but is otherwise stable and does not produce any random crashes.

Usability Features

When the game is loaded up for the first time, this help box is displayed to the user:

[Figure: First-run help box]

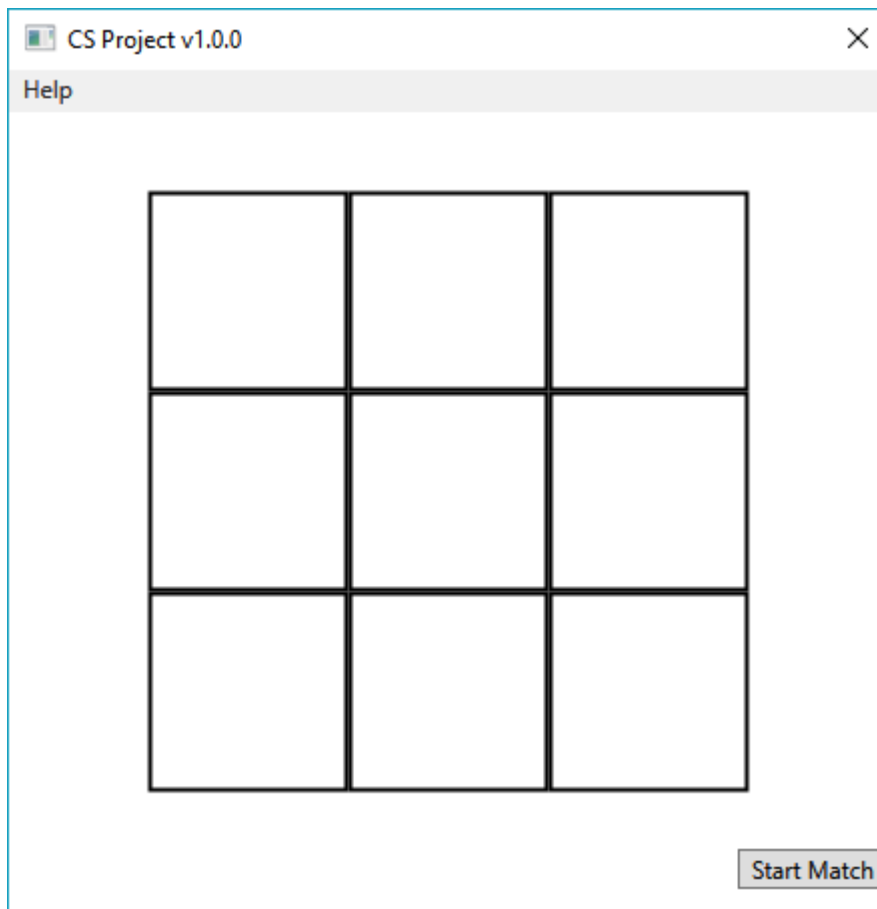


This is included in case the user does not know how to play tic-tac-toe. One improvement I could have made was to explain that to place pieces onto the grid, the user has to left click one of the slots on the grid.

This message box is only shown automatically one time (unless a user deletes a certain file the game uses). This is to stop it from being annoying to a user who has seen the message box many times before, but also brings it to the attention of any new user of the program.

After the user clicks 'Ok', they are greeted with the GUI looking like this:

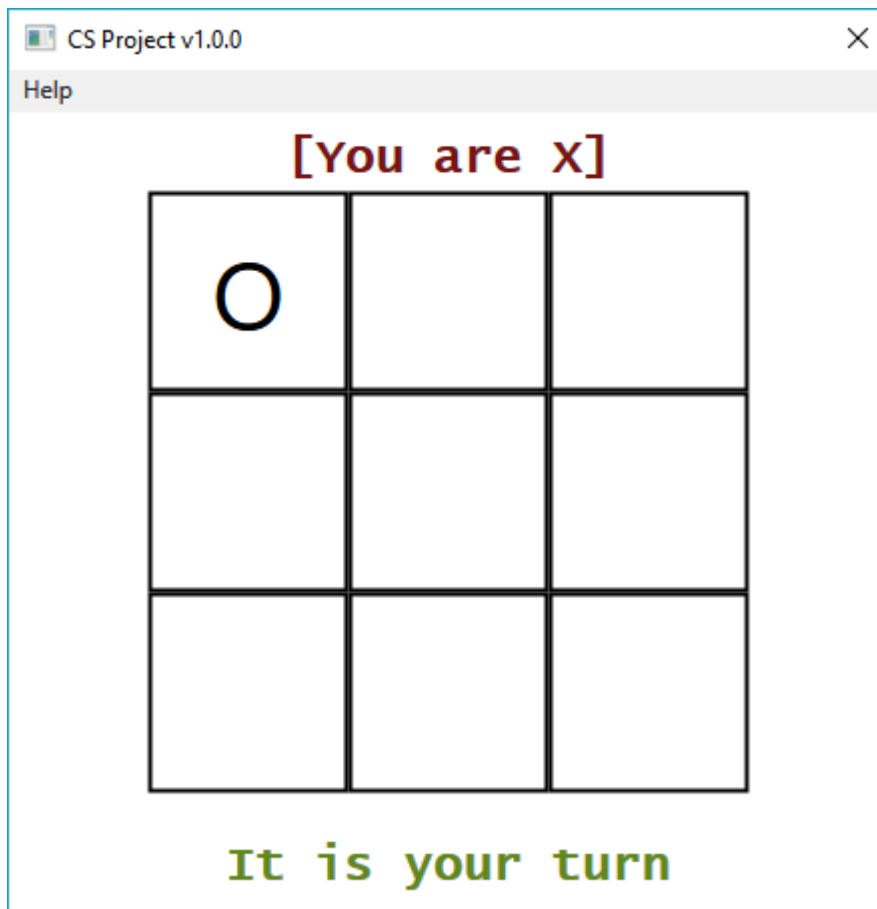
[Figure: Pre-game GUI]



The "Help" button at the top-left simply displays the help box again (so the user can bring it back up anytime, if needed). The only other usable button on the screen is the 'Start Match' button. No other information is displayed on screen at this moment, mostly because there is nothing else to display, and also so it doesn't overwhelm a new user of the program with multiple things at once. It also places focus on the 'Start Match' button.

After the user clicks 'Start Match', then the GUI will display some more information, and the AI will perform its first move:

[Figure: The GUI after starting a match]



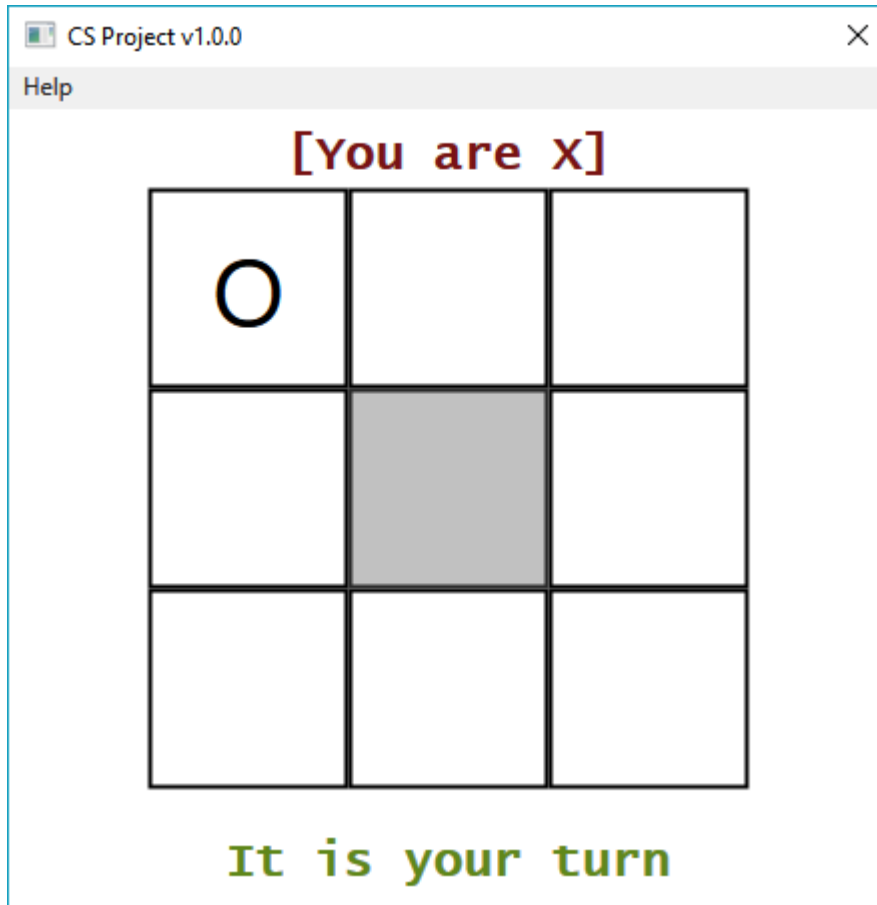
First of all, the top line of text informs the user that they are playing as the 'X' piece. This is to avoid the confusion of "who am I playing as?".

At the bottom of the screen, the game displays whether the AI or the player is taking their turn. While the game *does* change the text to say that the AI is taking its turn, because the AI makes its move so quickly, it usually flashes on screen for less than a second before it changes back to "It is your turn". This is of course included to make sure that the user isn't confused about who is taking their turn.

And finally, the 'Start Match' button is hidden as it is no longer needed until the match ends, meaning it leaves less clutter on the screen for the user to focus on.

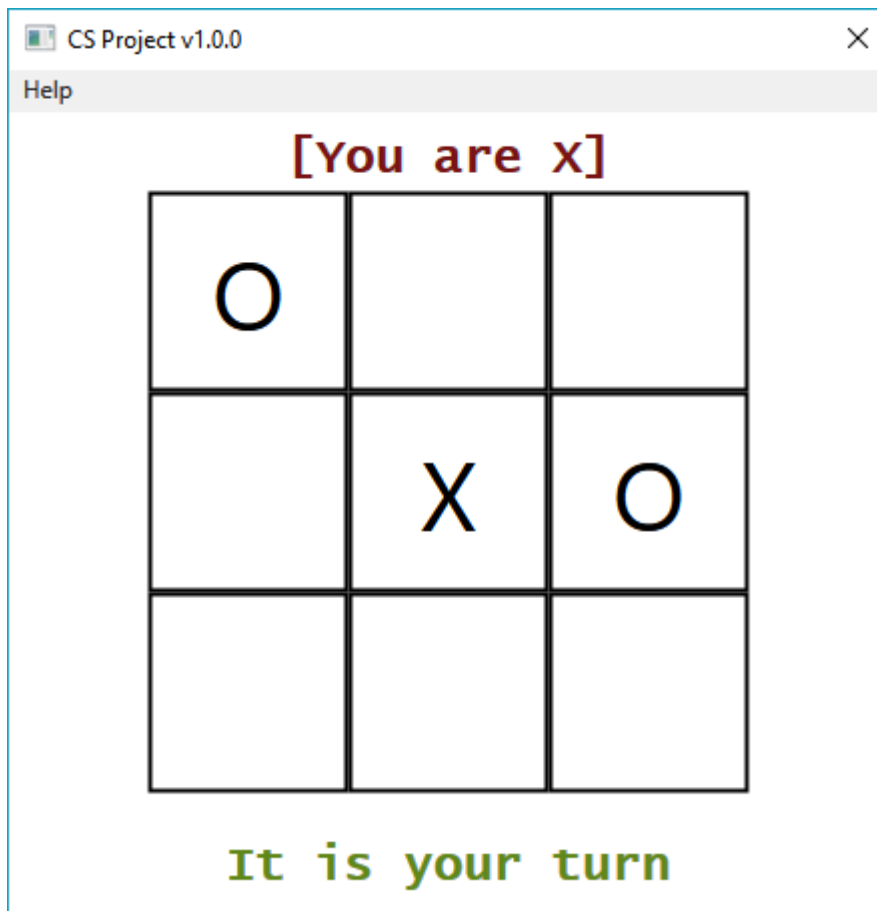
Next up is the 3x3 grid. Each slot in the grid will change the colour of its background whenever the mouse hovers over it, making it feel more interactive, and making it clear that the user can interact with the slots.

[Figure: The mouse hovering over a slot]



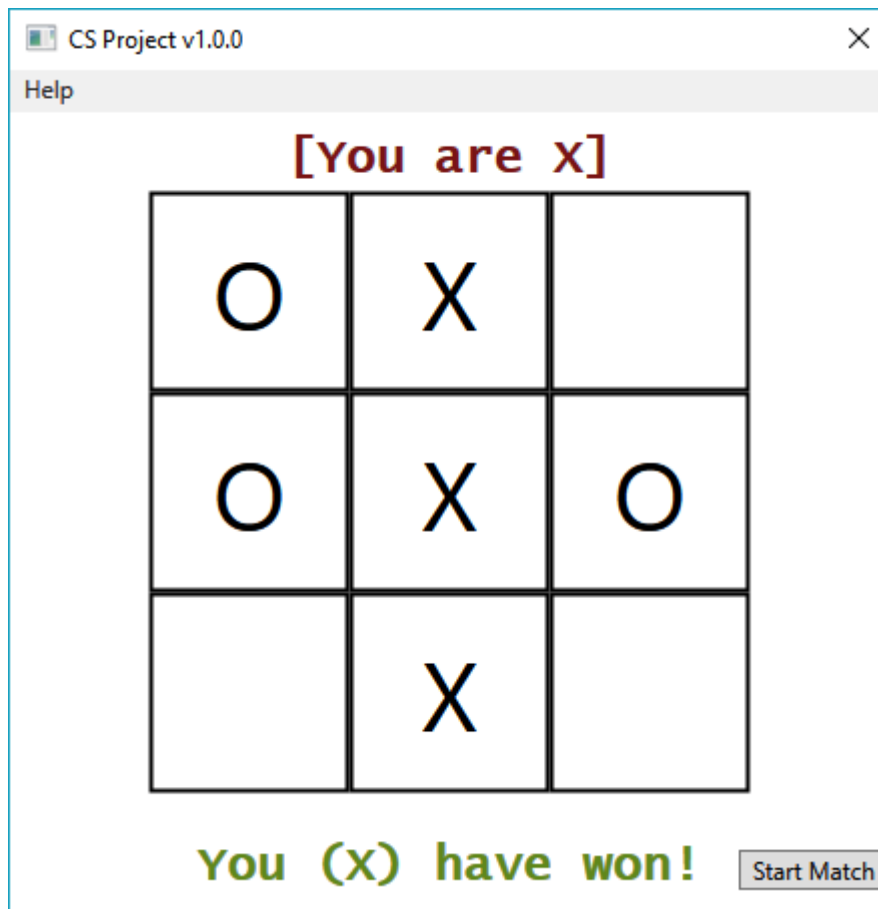
After the user clicks on a slot, then their piece will be placed, and the 3x3 grid will instantly update to display their move.

[Figure: The GUI after the user placed their piece]



Finally, at the end of a match, the 'Start Match' button is made visible again, and the game displays who won (or if there was a tie).

[Figure: The GUI after the player wins]

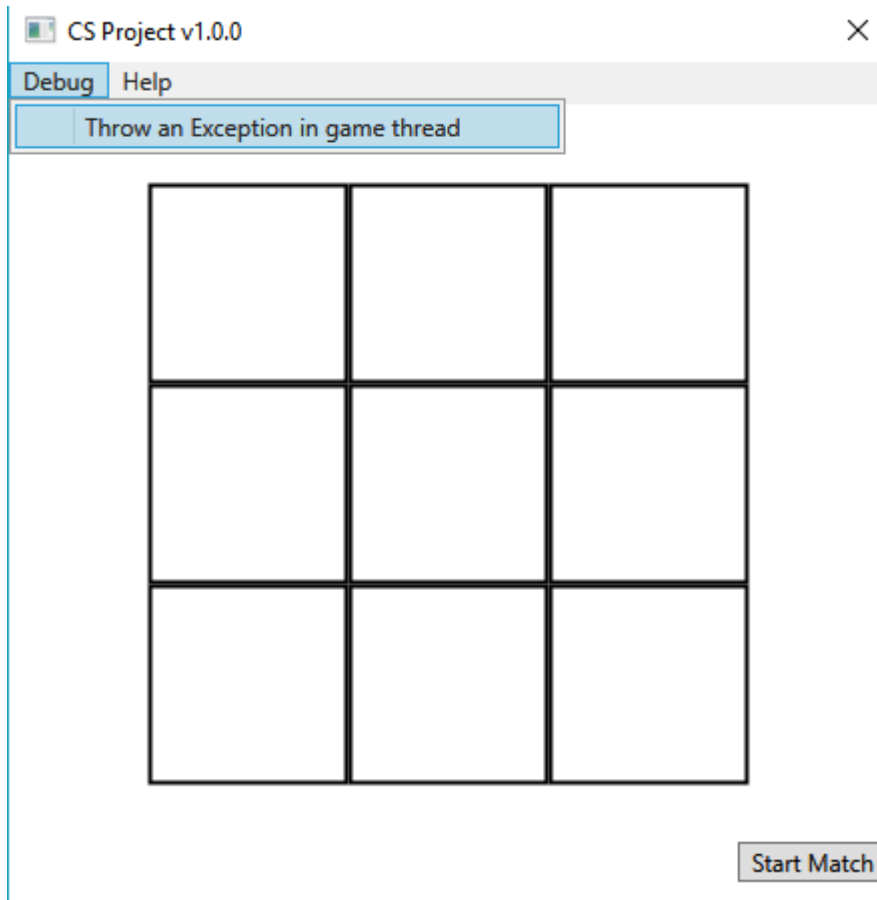


When my friends were testing my program, they had no issues figuring out how to start a match, and that they had to click a slot on the grid to place a piece, which possibly confirms that the GUI succeeds at being easy to use. The only issue they had was with the issue about the 'Start Match' button covering some text.

I will agree that I probably could have resized the window slightly, so I could place the 'Start Match' button somewhere where it didn't block text, as this makes it fail as a usability feature (and as previously complained to me, makes the GUI feel incomplete).

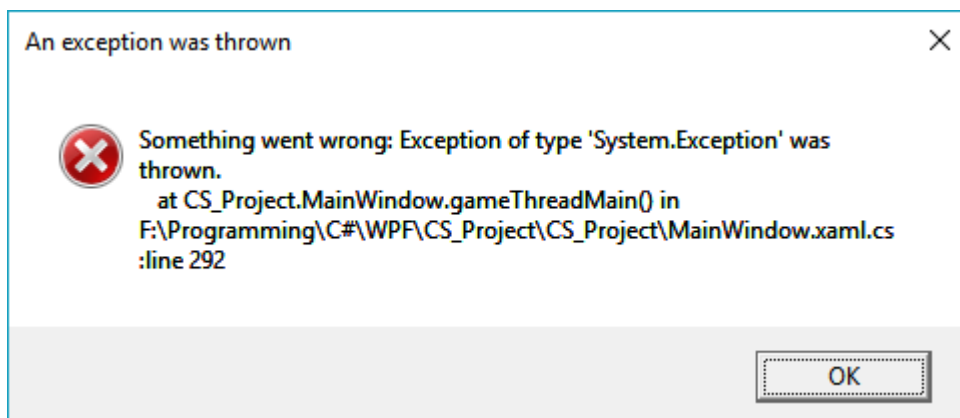
The GUI however, also features some elements to aid with development, that are only enabled if the project is built in a 'Debug' configuration.

[Figure: Debugging menu at the top left]



After using the debug option to throw an exception, this error box will also display.

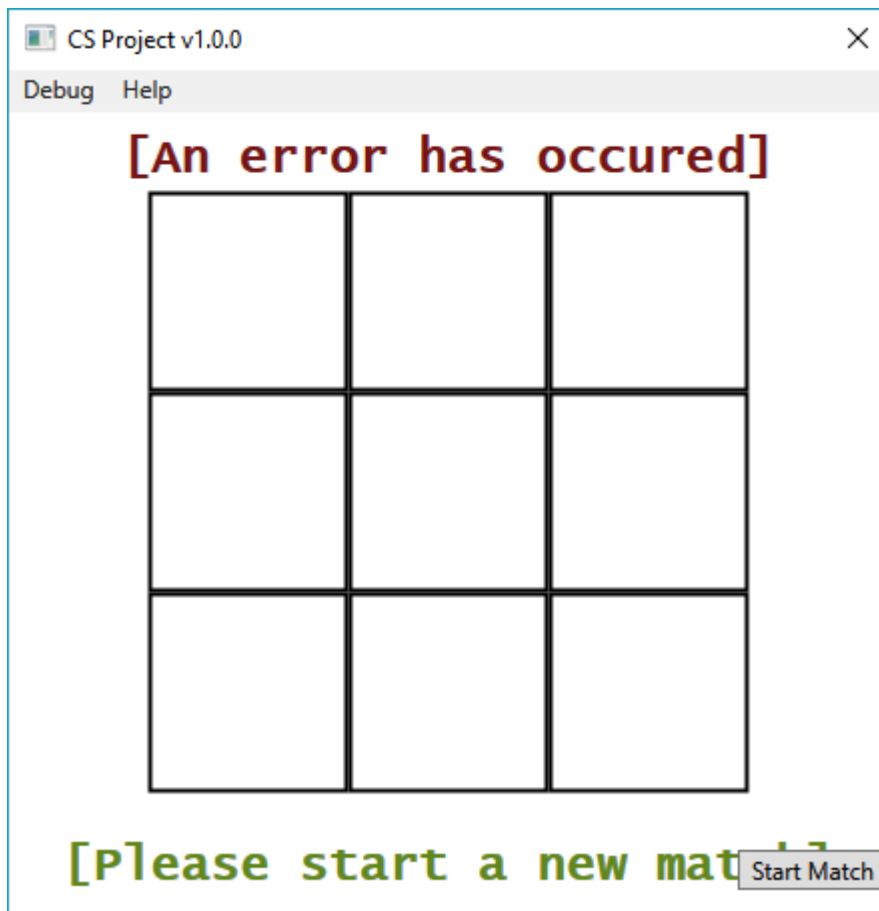
[Figure: GUI Error box]



A non-debug version of the game will also show this message box anytime an exception is thrown in the game thread, except without the stack trace. However, the only time an exception is thrown in the game right now is the 'data' folder bug, which because of unrelated reasons with WPF simply makes it so the game doesn't even open. The exception also wouldn't be thrown in the game thread, or in a try-catch statement, so this message box wouldn't appear either way in the case of that bug.

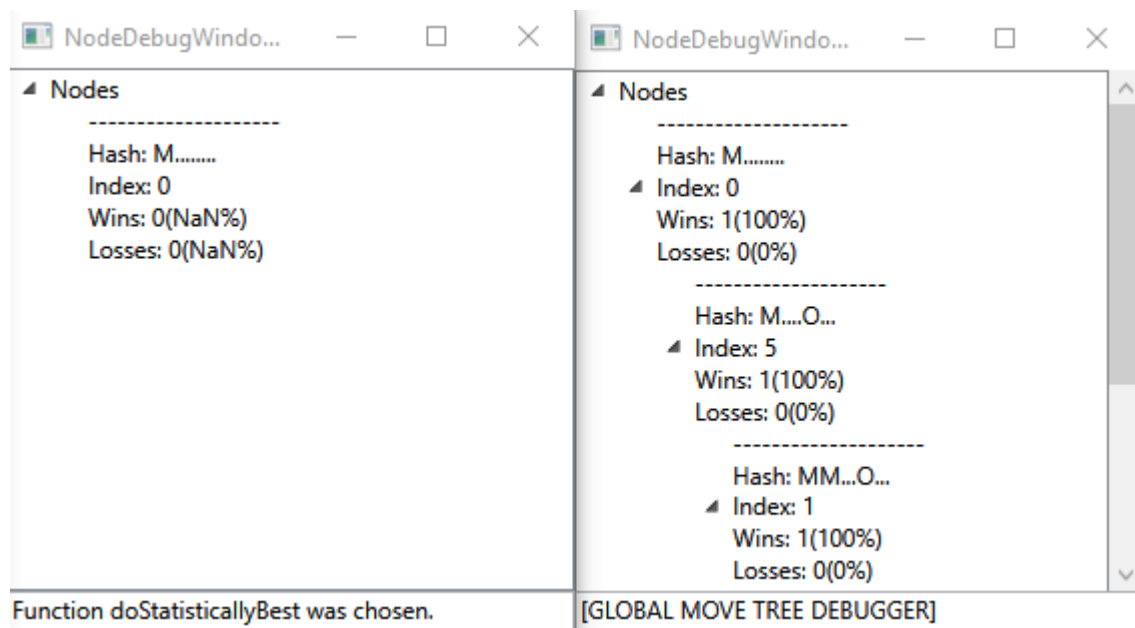
After this message box appears, the GUI will change some text on screen to inform the user of what to do:

[Figure: The GUI after an exception is caught]



The last usability feature the GUI provides, is when a match is started in a debug build of the game, it opens two special windows which visually display a tree of nodes:

[Figure: Node Debug Windows]



The left-hand window displays the AI's Local tree, and the right-hand window displays the AI's Global tree. I used these windows during development to ensure that the AI was selecting the correct moves, that it was keeping track of the local tree properly, etc.

Overall evaluation

Overall, the program works as expected, and if the 'data' folder bug, and the 'Start Match' button were fixed, it would satisfy all of the success criteria. The AI works as expected, although with some flaws in the algorithms it uses (as explained earlier, when the AI starts in only a single position).

While the sample size is small (4 people, including me) it has been user-tested, and there is agreement that the GUI is simple and easy to use.

Future Maintenance

While a decent chunk of the code is (in my biased opinion) well-made and structured, the rest of the code is rather lacking in areas.

First of all, I would like to re-explain that, because of the Board-controller architecture (or rather, just a basic object-oriented structure) it will be very easy for me to, in the future, create new controllers for the game. Examples include a controller for playing against another player over a connection, or another AI which uses a different algorithm (For example, the Minimax algorithm described in the Analysis). Once a new controller is created, it's as easy as going "Start a Match between PlayerController and MultiplayerController" to easily start a match between any two controllers.

Next, I would like to talk about the `MainWindow` class. The `MainWindow` class contains code for when the user interacts with the GUI (and therefore, the code for sending messages to the game thread), code to start up the game thread, the message queue, and the game thread's main function. This makes the `MainWindow` class messy, and it contains code not entirely relevant to the window, so in the future one of the first things I may do is to move the message queue, and the game thread's main function into a `'GameThread'` class. This would separate the game thread's logic code from the `MainWindow` class, and would allow me to create unit tests that make use of the game thread, since currently it would require me to make a `MainWindow` instance so the test could access the message queue.

Onto the `Board` class. For the most part it's fine, however the `'startMatch'` function does not contain any code to handle when a controller throws an exception, which means that if an exception is thrown, then the `'Board'` class doesn't run some of its clean-up code which leaves it in an invalid state, requiring a new instance of `Board` to be created, which should be fixed in the future. Secondly, the `'checkWin'` function could easily (and should) be moved into the `Hash` class, so controllers can easily predict a hash of the board for if they did a certain move (or certain sets of moves) and then use the `'checkWin'` function to see what the result of the move(s) would be.

The two issues preventing the project from completely meeting its success criteria, the `'data'` folder bug and the `'Start Match'` button GUI issue, should be fixed as soon as possible.

As explained previously, when the `'StatisticallyBest'` algorithm is used with the `'WalkPaths'` algorithm, they produce an $O(2n)$ (I've also explained that this is an inaccurate Big-O notation) algorithm. I have thought about an algorithm which seems to be $O(n)$, where n is the number of nodes in the tree, however this would require ditching the generic usability of the `'WalkPaths'` algorithm (by merging it into `'StatisticallyBest'`) while also rewriting the `'StatisticallyBest'` algorithm, ultimately allowing the algorithm to only look over each node a single time. This is undesirable to me, so unless I give the project any further attention to the point where the AI is working with a large amount of data, I won't be performing this change.

Finally, the existing unit tests should be updated, and new tests should be created (especially when the "Move code from `'MainWindow'` into `'GameThread'` class" change is completed) because as I have explained several times in this document, a few bugs got past me that should've easily been caught by tests, if they were thought through properly.

Roadmap

The roadmap is used to plan out (and log) features for each tagged version of the game. Git tags will be used as well so a simple `git checkout tags/v0.1.0` can be used to check what the code was like at a certain point.

The format of this roadmap is that there is first a major version “v0.X.0” defined alongside goals planned for the major version to be reached. Following this definition is a log of minor versions “v0.X.Y” (where ‘Y’ is the minor version) detailing what each minor version added (with the ultimate goal of fulfilling the goals of the major version), a screenshot of the tests for the version, and a screenshot of the GUI.

V0.1.0

This version should have the raw foundation of the game, including the GUI for the player, the class to represent a game board, a few pre-requisites for AI, and a way to represent a tree of moves.

Goals:

- Have a class to represent the game board [V0.0.2]
- Have a basic controller class setup for the Player (AI will come later) to interact with the board via a GUI [V0.0.3]
- The GUI should at the moment, only bother displaying the game board, as well as who's turn it is [V0.0.3]
- The game board class should have the basic game rules implemented (can only place pieces on empty spaces. Game is won if 3 pieces are in a row, etc.) [V0.0.2]
- Have a way to hash the game board [V0.0.1]
- Have a way to represent a tree of moves (the Node class and the MoveTree class provide this) [V0.0.1]
- Have a way to calculate the statistically best path of moves to make (Average.statisticallyBest sorts this out, I know it's a bit strange to have this before working on the AI, but I just needed it out the way) [V0.0.1]
- Allow any Node class to be serialised/unserialised. [V0.0.1]

Final version for V0.0.X

The final version of every Minor version update should contain general clean-ups/refactoring of code. The final version is generally also the next minor version. So the version 0.0.4 is also v0.1

Goals:

- Remove the 'MoveTree' class. First, create a Node.root property to create an empty node (suitable as a root). Then, move the MoveTree.walk function into the Node class, since it makes more sense and provides more flexibility there. [Done in V0.0.2]
- Use the 'Board' class as a namespace for the 'Hash' class. So 'Hash' becomes 'Board.Hash'. This better reflects its purpose, and becomes consistent with 'Board.Piece'. [Done in V0.0.4]
- ~~Change 'Node' and 'Hash' to be created using [Object Initialisers](#), as it creates cleaner (and clearer) code.~~ (After a change of thought, this could be more of a burden, as it would require me to make some read-only properties non-read-only, and would make lines look bloated)

V0.0.1

Achieved:

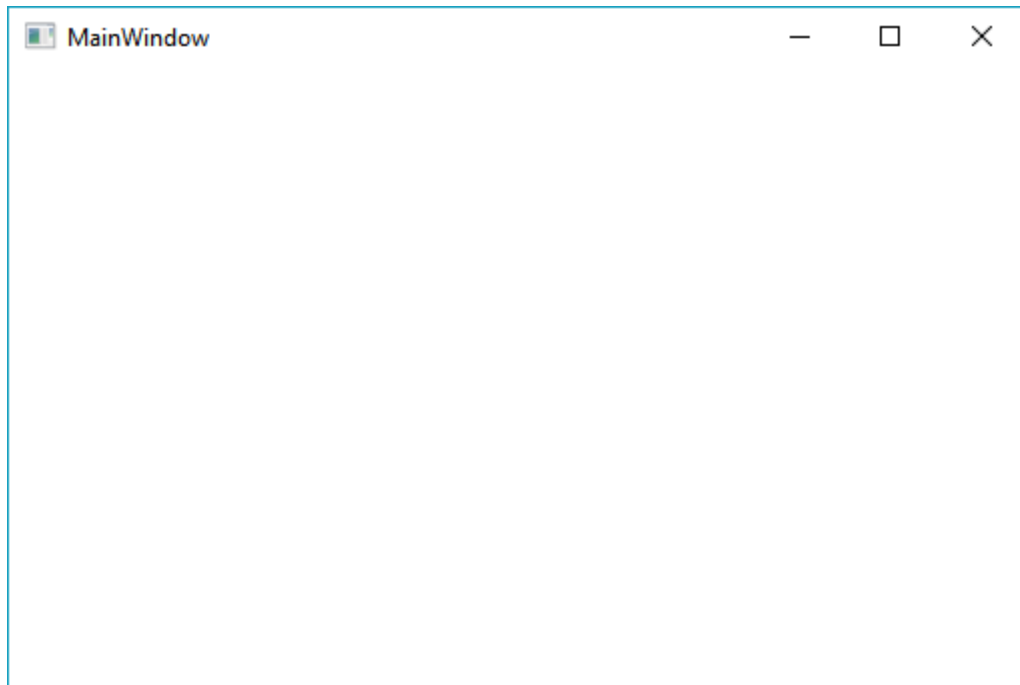
- Have a way to hash the game board
- Have a way to represent a tree of moves
- Have a way to calculate the statistically best path to make
- Included this document

Tests:



Passed Tests (3)		
✓	statisticallyBestTest	4 ms
✓	testHash	88 ms
✓	walkTest	2 ms

Screenshot:



This version added some very core features to the game. It implemented the Hash of a game board, a way to represent a tree of moves, and an implementation of the 'statisticallyBest' algorithm.

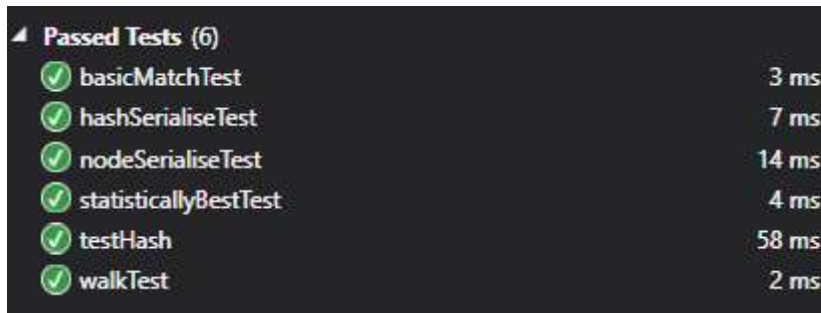
I decided to add in these features very early on so I could begin to focus mostly on getting the GUI, Board (meaning the game loop and game rules as well), and a PlayerController setup.

V0.0.2

Achieved:

- Add basic serialisation support.
- Add the 'Board', and 'Controller' classes, with very basic tests. (This was a lot of code)
- Implement serialisation for the 'Hash' and 'Node' class.
- Remove the 'MoveTree' class, and move its functionality to 'Node'.

Tests:



Passed Tests (6)	
✓ basicMatchTest	3 ms
✓ hashSerialiseTest	7 ms
✓ nodeSerialiseTest	14 ms
✓ statisticallyBestTest	4 ms
✓ testHash	58 ms
✓ walkTest	2 ms

Screenshot: [Exactly the same as V0.0.1]

MoveTree was removed because it wasn't offering any special functionality that I couldn't just put inside 'Node'. A special 'Node.root' function was made to replace usage of 'MoveTree.root'.

Moving the functionality into the 'Node' class also allows any single 'Node' to function as the root of a tree, allowing much more flexibility.

V0.0.3

Achieved:

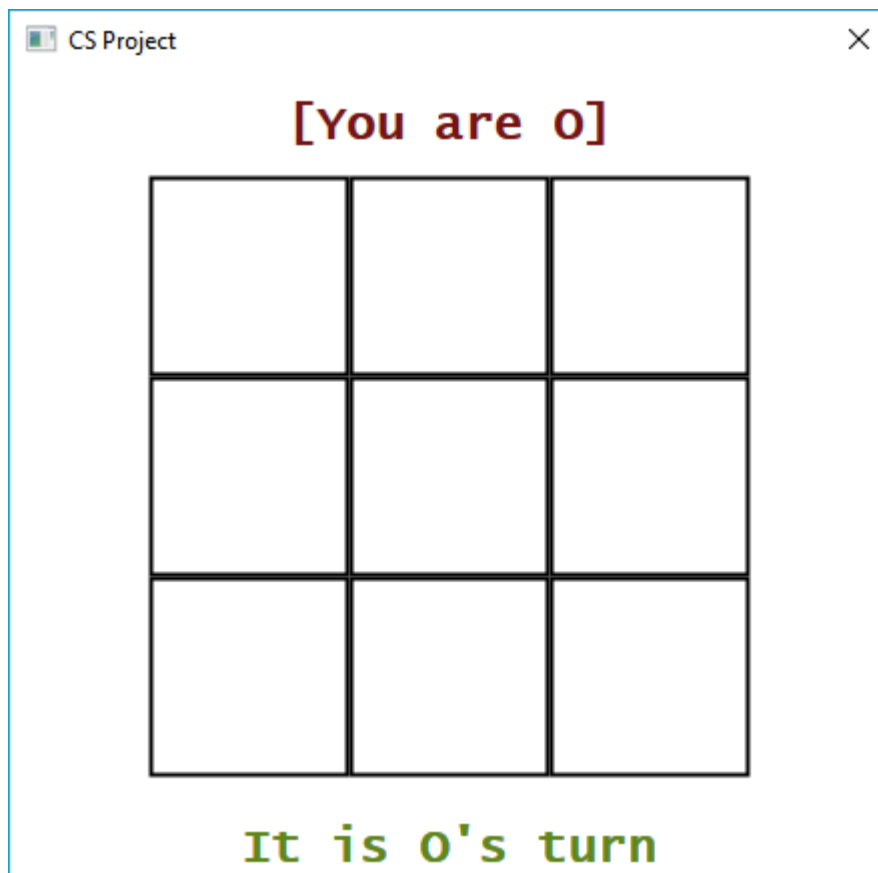
- Add support for running a thread for all the game logic, and providing a way for the two threads to speak to each other.
- Add a GUI, and a controller to let the player interact with the board.
- Add code to handle when two controllers tie.
- Fixed a bug in the win checking code. It was checking a wrong slot when checking “Top right to bottom right”.

Tests:

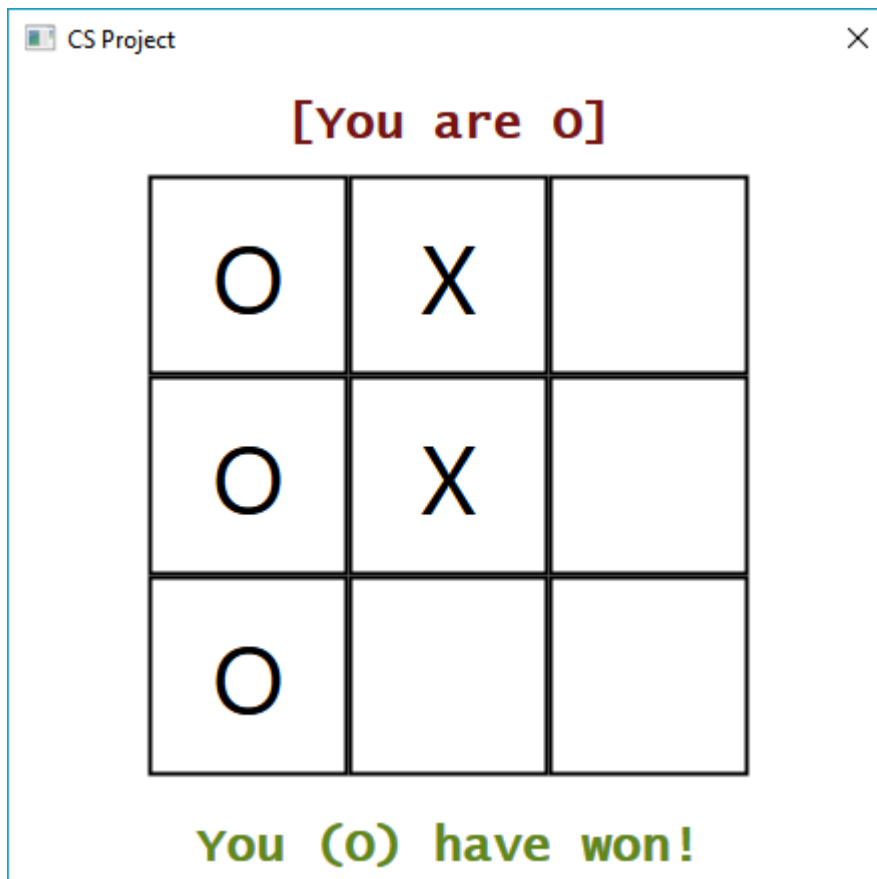
Passed Tests (6)		
✓	basicMatchTest	7 ms
✓	hashSerialiseTest	15 ms
✓	nodeSerialiseTest	6 ms
✓	statisticallyBestTest	3 ms
✓	testHash	48 ms
✓	walkTest	1 ms

Screenshot:

[The GUI just from opening the program]



[The GUI after one of the players have won]



When deciding how the game loop should work, there were 3 different ideas that came to mind.

The first idea was to run the game logic in the same thread as the GUI. However, with the way the game loop works and is structured, it would cause the GUI to freeze and prevent the user from actually interacting with the game. This obviously was not ideal.

The second idea, and the one I went with, was to use a separate thread for the game logic, and provide a way (in this case, a thread-safe queue) to allow the GUI thread to communicate with the game thread.

The final idea was using C#'s support for asynchronous tasks. However, in my past experience with asynchronous code (In C#, at least), I haven't ever been able to structure things in a sane way, and usually end up with a buggy, unreadable mess. So, I decided to avoid it.

As a side note, in this version of the game, two PlayerControllers are put up against each other, so it is at this point a player vs player game.

V0.0.4

Achieved:

- Put the 'Hash' class under the 'Board' namespace, so it is used like 'Board.Hash'.

Tests: [Same as V0.0.3, as no major change was made]

Screenshots: [Same as V0.0.3, as no major change was made]

My reason behind this decision is because, in my eyes, the code is more structured like this.

V0.2.0

This version should be an unpolished version of what the final game will be like. This version should include at least 1 AI mode, a move tree which the AI uses to decide its move (and a way to load/save this data to a file), and ~~better management for when the game thread throws an exception.~~

Goals:

- Have at least 1 controller for the AI, which should use the 'Average.statisticallyBest' function to determine its next move. [V0.1.2]
- Create a static class which contains the code for interacting with the file system. The reason for a separate class is to allow a cleaner interface for the rest of the code, as well as separating file input/output code from things such as the AI. Example = `'var move_tree = GameFiles.loadMoveTree(MoveTrees.Global);'` to load the global move tree, the AI doesn't need to know the specifics of it such as the file name, so the GameFiles class is used as an abstraction. [V0.1.3]
- ~~Provide a function inside of MainWindow for the game thread to call whenever an unhandled exception occurs (this function is called using the window's dispatcher). The game should not crash due to its own mistakes, so any exception thrown must be reported and fixed. The game is only allowed to crash if, for example, the user decides to corrupt one of the files used by the game, and the game chokes when loading it. In short, crash due to the user, not due to buggy code. But even then, it's ideal to just display an error message instead of crashing, since the program should still be in a valid state from a user error. [Never mind, seems the program will crash regardless of what thread throws it. Which is the desired behaviour.] [V0.1.3 sort of implements this though]~~
- Modify the 'Node.walk' function so it will instead, now allow a function to be called on every node that is walked to. The old functionality can still be replicated because the new functionality will allow much more flexible code, and will allow new opportunities. [V0.1.1]
- Provide an easy to use interface in the class described in the 2nd bullet point to save/load arbitrary trees of nodes.
Possible api = `"GameFiles.saveTree(myRootNode, "super_tree");",`
`"var root = GameFiles.loadTree("super_tree")"` [V0.1.3]
- Provide a way to merge the data from one move tree into another. For example, say I gave my friends the current version of my project to play with, and then I asked them for their AI move trees so I could merge them all into my personal AI move tree. This would allow for the AI to make use of more data. The algorithm the AI uses in onMatchEnd could be turned into a function for this. [V0.1.3]
- Provide a separate debug window that an AI can hook up into to allow a Graphical representation of their move tree to be seen, as well as tools to modify the tree. This window can also be used so the AI can report back any info I need. [V0.1.2, no functionality to edit the node tree]
- Using the algorithm used in "Average.statisticallyBest", create a function such as 'Node.walkEveryPath' which will go over every possible path found in the tree, and perform an 'Action<List<Node>>' on every path found. [V0.1.1]

List of bugs that need fixing:

Description	Status
When a match isn't being performed, the player can queue up moves by pressing any of the slots. This will create a backlog of PlayerSelectMessages. Fixing it should be as easy as clearing the message queue anytime the player controller's onDoTurn function is called.	Fixed in 0.1.4
[Nothing will be done about this, unless I have time after everything else. The AI as it stands suffices for the project.] The AI, because it lacks enough data, will always start in the same position when the 'doStatisticallyBest' function is being used. I'm not too sure how to remedy this as of now...	Won't be fixed
If an exception is thrown in the game thread, then the game thread will end early and the UI will effectively be doing nothing. This will just require changing where the try-catch is, as well as resetting some variables.	Fixed in 0.1.4
The exception thrown in the Game thread, which is used to stop the thread when the window is closed, is passed into MainWindow.reportException. So, an error box appears every time the window is closed. This will simply require checking what exception is thrown, and preventing the one we don't want from being passed.	Fixed in 0.1.4
When the AI wins a round, the GUI doesn't get updated to show where the AI put their winning piece. I'm pretty sure a single line of code will fix this.	Fixed in 0.1.4

V0.1.1

Achieved:

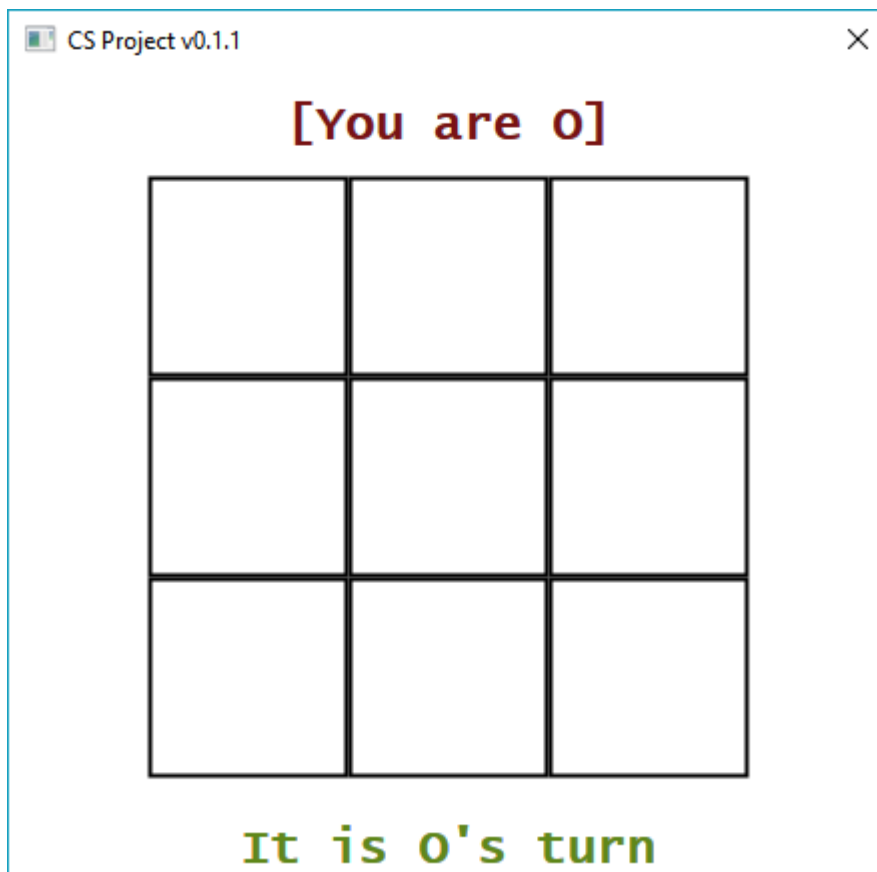
- Display the current version of the project within the window's title. E.g. "CS Project v0.1.1 prototype". If "prototype" is displayed, then it is an incomplete version. So "v0.1.1 prototype" is an incomplete version of version 0.1.1, whereas "v0.1.1" would be the final, complete version of version v0.1.1.
- Create 'Node.walkEveryPath' which is a function based off of the algorithm used in 'Average.statisticallyBest'. This function allows code to perform an action on every possible path in the tree. 'Average.statisticallyBest' was modified to use this new function.
- Modify 'Node.walk' to allow an action to be performed on every node walked to (tests were update as well).

Tests:

Passed Tests (6)		
✓	basicMatchTest	7 ms
✓	hashSerialiseTest	67 ms
✓	nodeSerialiseTest	6 ms
✓	statisticallyBestTest	5 ms
✓	testHash	73 ms
✓	walkTest	6 ms

Screenshot:

[The GUI now shows the version of the project in the title bar]



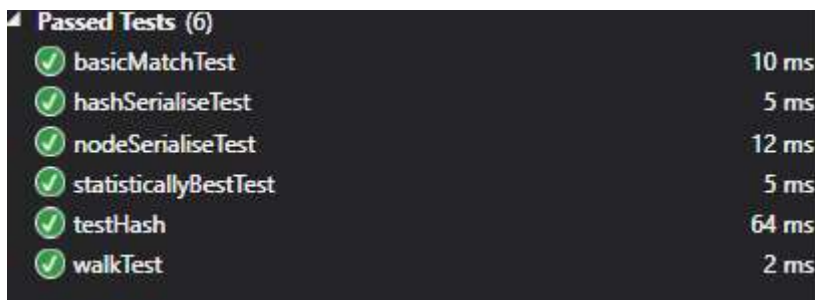
This version of the game will not have a single noticeable difference when playing the game, since this is exclusively backend stuff (that for the most part, isn't even used yet). It focuses more on making it easier to write code that traverses a node tree, which will be very beneficial for the near future for the AI/any other algorithm that uses the node tree.

V0.1.2

Achieved:

- Create Config.versionString, which stores a string representing the current version of the game. Every window will have this string appended to its title.
- Add a Debug window which can be used to visually see a tree of nodes.
- Add a 'Start Match' button which will start a new match between the AI and the player.
- Modify the Controller base class so that onAfterTurn and onMatchEnd are given the current state of the board, and the index of the last piece placed. This is so the controllers don't have to track this information themselves, which creates bugs. It also means they can be certain the data is correct.
- Add an 'AI' controller, which currently has a 'doRandom' and 'doStatisticallyBest' mode.

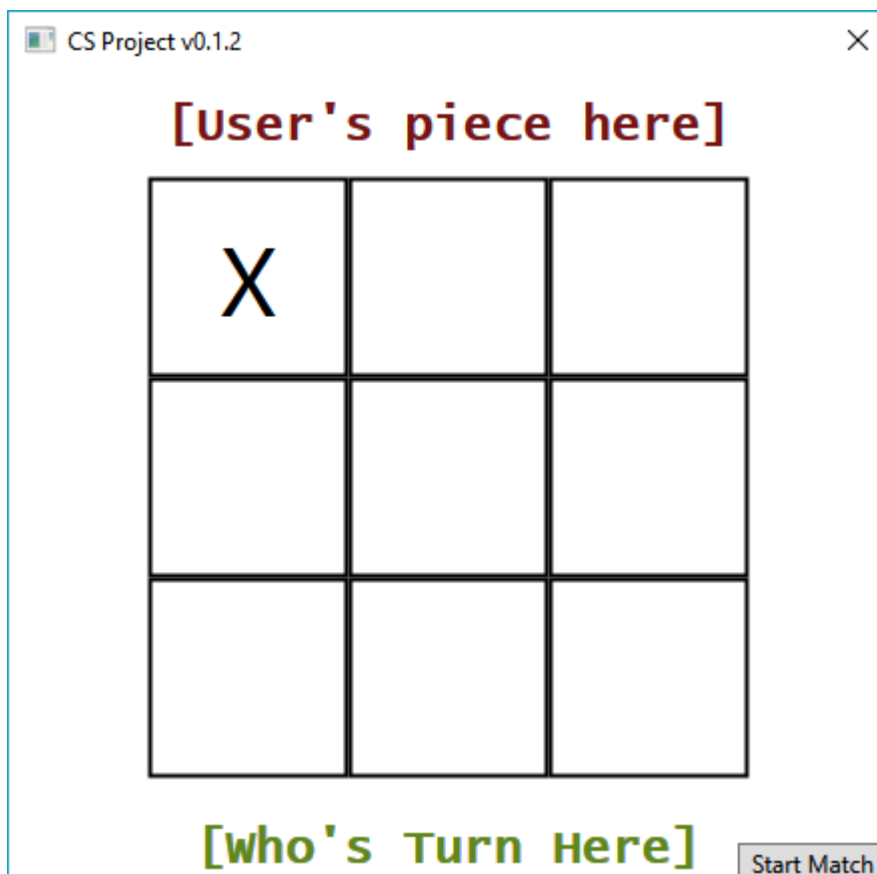
Tests:



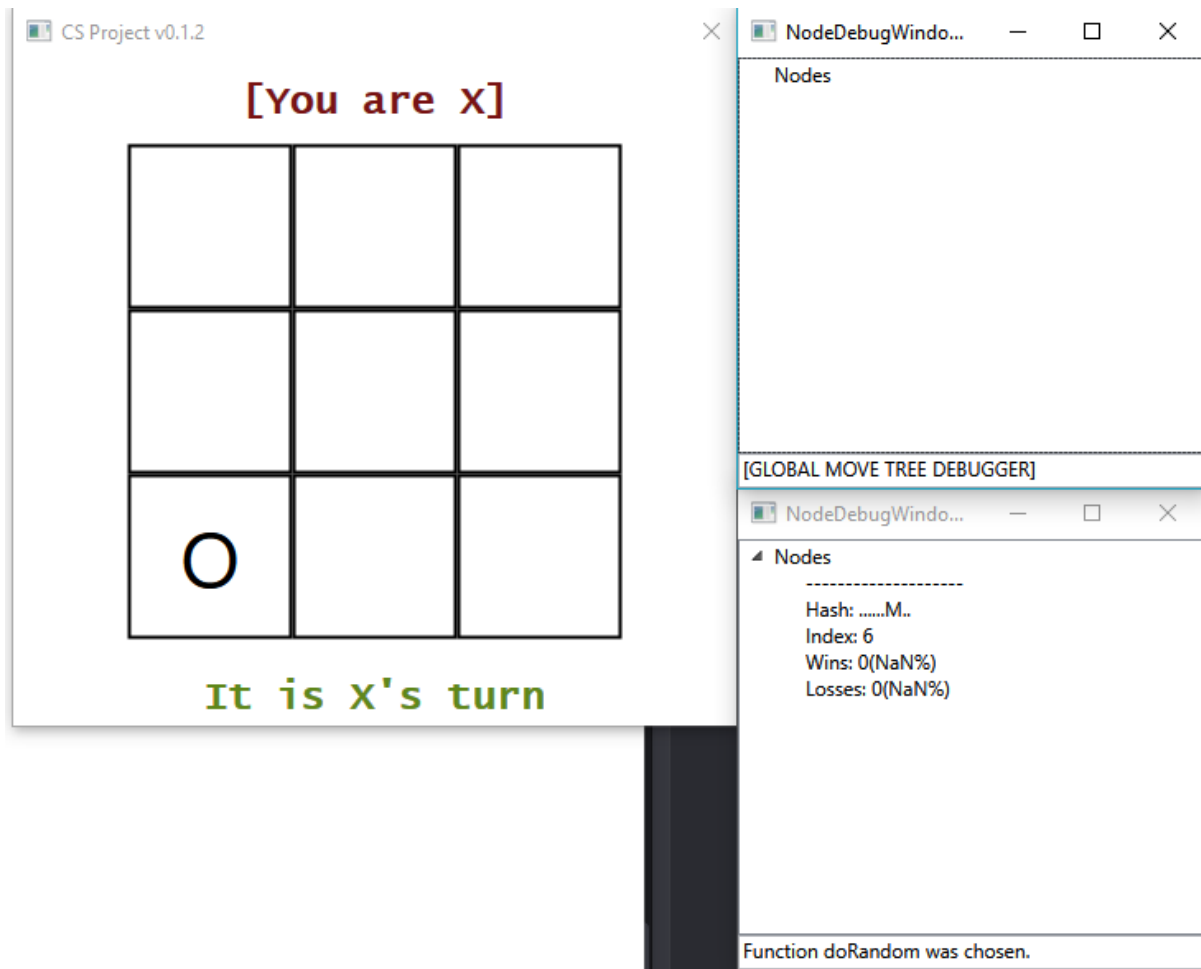
Passed Tests (6)		
✓	basicMatchTest	10 ms
✓	hashSerialiseTest	5 ms
✓	nodeSerialiseTest	12 ms
✓	statisticallyBestTest	5 ms
✓	testHash	64 ms
✓	walkTest	2 ms

Screenshot:

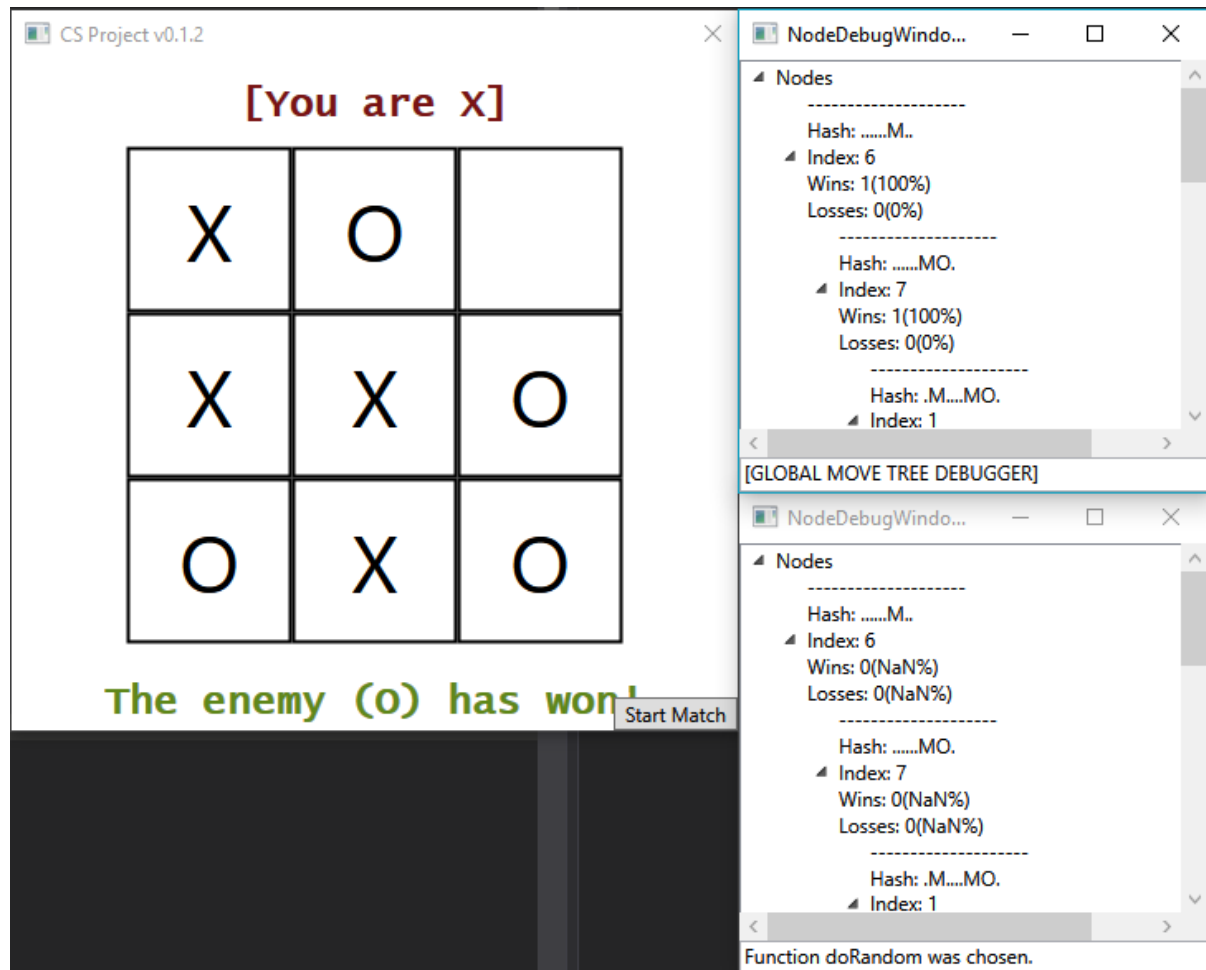
[The GUI, which now features a 'Start Match' button]



[After the 'Start Match' button is pressed, the debug windows open. The top debug window is for the global tree, while the bottom one is for the local tree.]



[At the end of the match, you can see that the local tree(bottom) has been merged into the global tree (top). Also, a bug can be seen, where the winning piece (placed by the AI in the top-right) doesn't actually get displayed.]



This update is the first on to feature the AI's main mode, 'doStatisticallyBest', in action. Currently the AI can't load data from previous sessions, so it has to remake its global tree from scratch each time the program is run.

The AI, once it has played one winning match, will always try to place its first piece in the exact same position as the first winning match. This is simply because there's not enough data for the AI to work with.

I will have to figure out a way to try and get the AI to randomise whether it uses 'doStatisticallyBest' one match, and 'doRandom' another match, so it has more data to work off of.

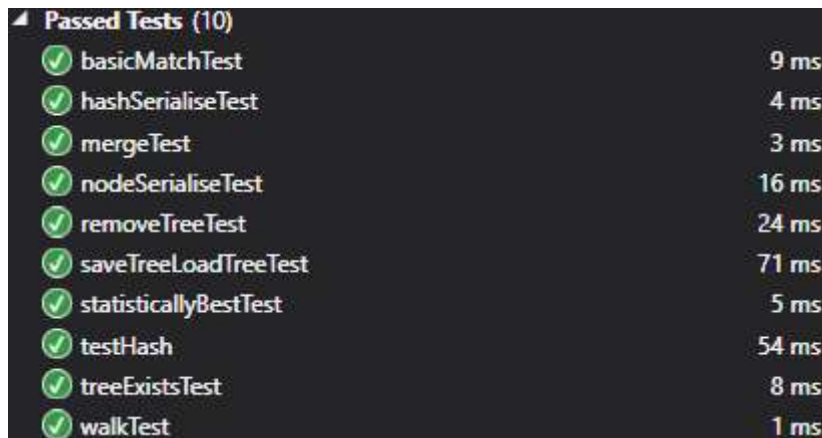
Another solution might be to pre-create a global tree, with one winning path for each starting position the AI could use. He should then have enough data to properly choose any of the slots to start off with.

V0.1.3

Achieved:

- Exceptions thrown in the game thread will trigger a message box to appear in the UI thread, detailing the exception. In debug mode, this message box contains a stack trace. Ideally, this should only ever be seen in cases of I/O issues (such as a game file being corrupt). Any other kind of exception should require a fix in the code to prevent it. Despite saying I wouldn't do this, it's more user-friendly to say something's gone wrong, rather than being silent and having the program not act correctly.
- Add the GameFiles class, which currently allows code to save and load trees by using names. `GameFiles.saveTree("MyTree", root) | var root = GameFiles.loadTree("MyTree")`
- Modify the AI to save and load its global move tree using the GameFiles class.
- Add Node.merge, which is a static function used to merge a 'source' node tree into a 'destination' node tree. The AI was originally using this algorithm to merge its local move tree into its global move tree. The original algorithm had no support for if the 'source' node tree had more than 1 path, so that support was added.

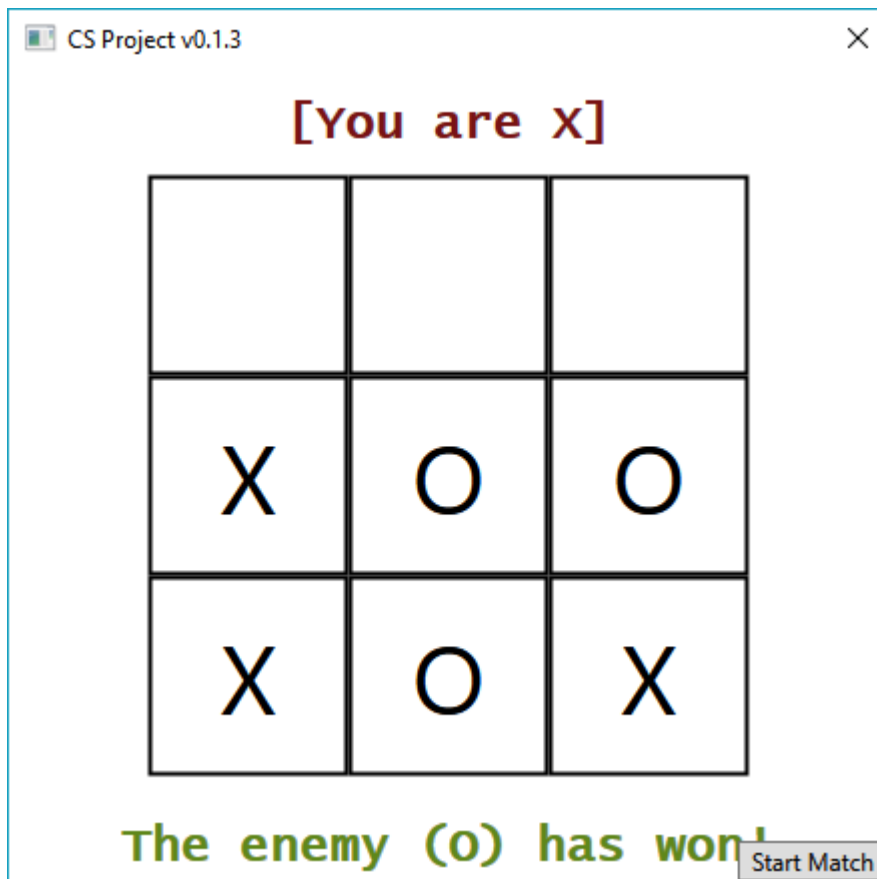
Tests:



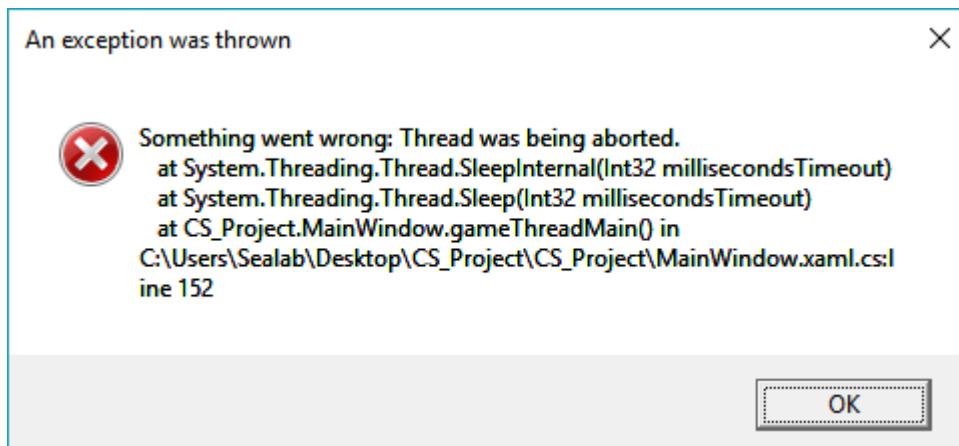
Passed Tests (10)	
✓ basicMatchTest	9 ms
✓ hashSerialiseTest	4 ms
✓ mergeTest	3 ms
✓ nodeSerialiseTest	16 ms
✓ removeTreeTest	24 ms
✓ saveTreeLoadTreeTest	71 ms
✓ statisticallyBestTest	5 ms
✓ testHash	54 ms
✓ treeExistsTest	8 ms
✓ walkTest	1 ms

Screenshot: [Next page]

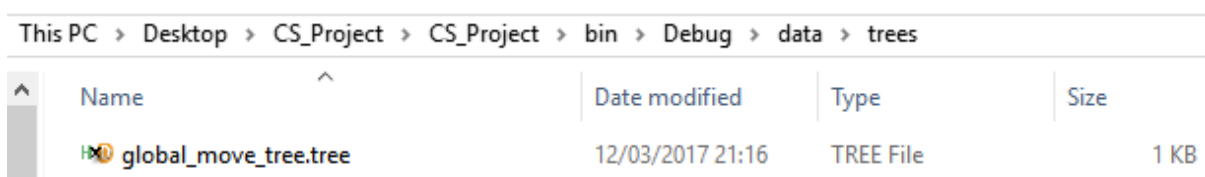
[The AI's winning piece still does not display]



[An example of the new exception code, thanks to a bug (well...) where an exception is thrown when the game thread is closed. In Release mode, no stack trace appears.]



[Proof that the global tree is saved to a file]



After this version, all that's left will be bug fixes and changes to the algorithm the AI uses. After those are done, then the UI/overall experience needs to be polished (V0.3.0 will have the details). Once that's all done, then the game is in a finished state (V1.0.0).

I was surprised (as well as happy) to find that the AI worked perfectly fine, without any issues, when I added the code to save/load. It makes me feel it's at the very least coded in a good way. It also confirms to me that GameFiles' interface is simple enough, as only 5 lines of code was needed (2 lines were calls to the GameFiles class, 2 were for error checking, 1 was adding a variable. Comments are not included in this number).

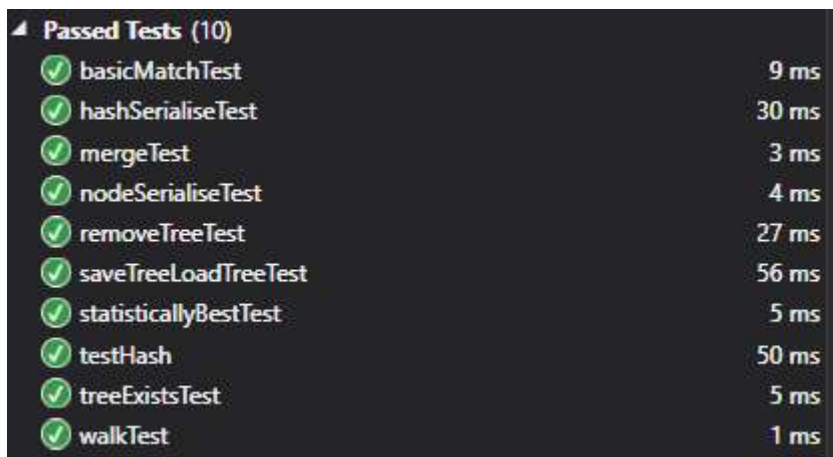
The debug windows + the unit tests for GameFiles makes me certain that all the data is being written/read in correctly.

V0.1.4

Achieved:

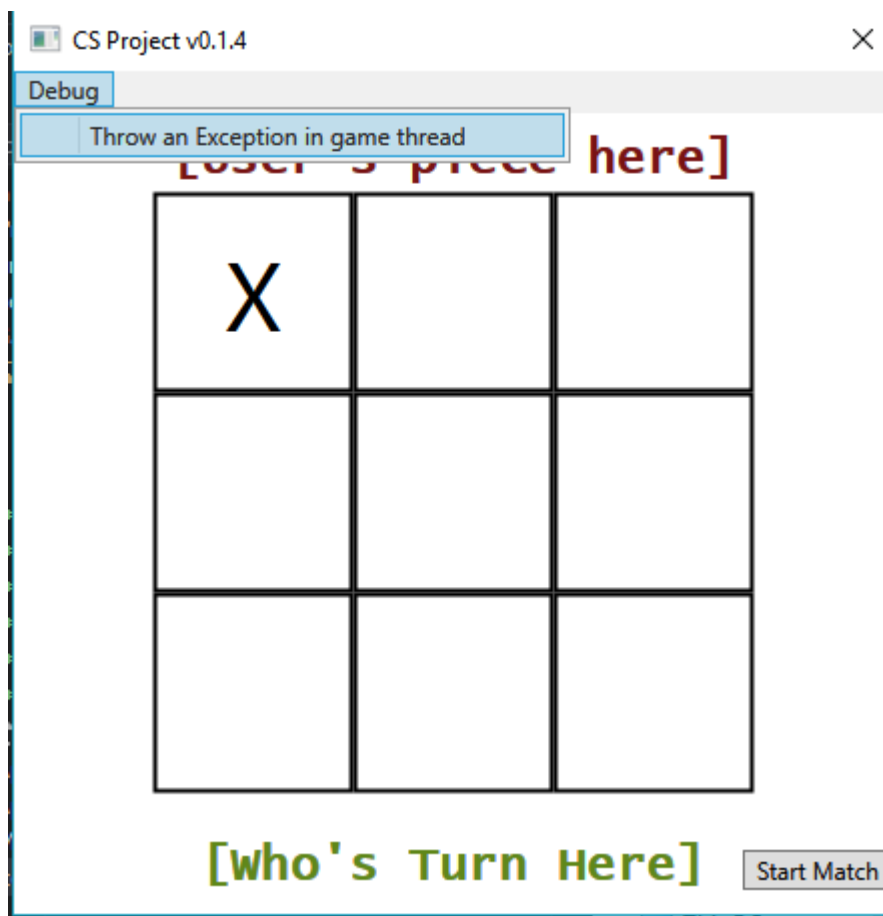
- Fixed a bug where, if the AI placed the last piece of the match, then the GUI wouldn't update to display where the AI put its piece.
- Fixed a bug where, if the player spam clicks empty slots on the 3x3 grid, it will queue up all of the moves and perform them one at a time.
- Add a toolbar at the top of the GUI. Currently the only menu to select is 'Debug', which contains debug functions relevant for testing.
- Add a debug function into the debug menu which causes the game thread to throw an exception. This is used to test how the game thread handles exceptions.
- Fixed a bug where, if the game thread threw an exception, then the game thread would be terminated; making the game unresponsive (it wouldn't crash, but it wouldn't do anything since the game thread won't be there to process things). (This is also where the debug menu is useful)
- Fixed a bug where an error box is shown anytime the game is closed.
- Add Board.predict, which is a function used to 'place' a piece on the board temporarily to 'predict' the result of the match. [This later turned out to be useless for what the issue was, so was removed]
- Add a new mode to the AI, "doWinBlockCheck", where the AI determines if it can either win this turn, or block the player from winning this turn. However... This mode has been disabled and won't be used by the AI(Explained below).
- If the AI selects a path that has less than 25% chance of winning, then there's a 25% chance of the AI using 'doRandom' for a single turn. This prevents the AI from going down paths that generally make it lose, without locking the path out complete from the AI's view.
- When the game isn't built in debug mode, prevent the node debug windows from being created.

Tests:

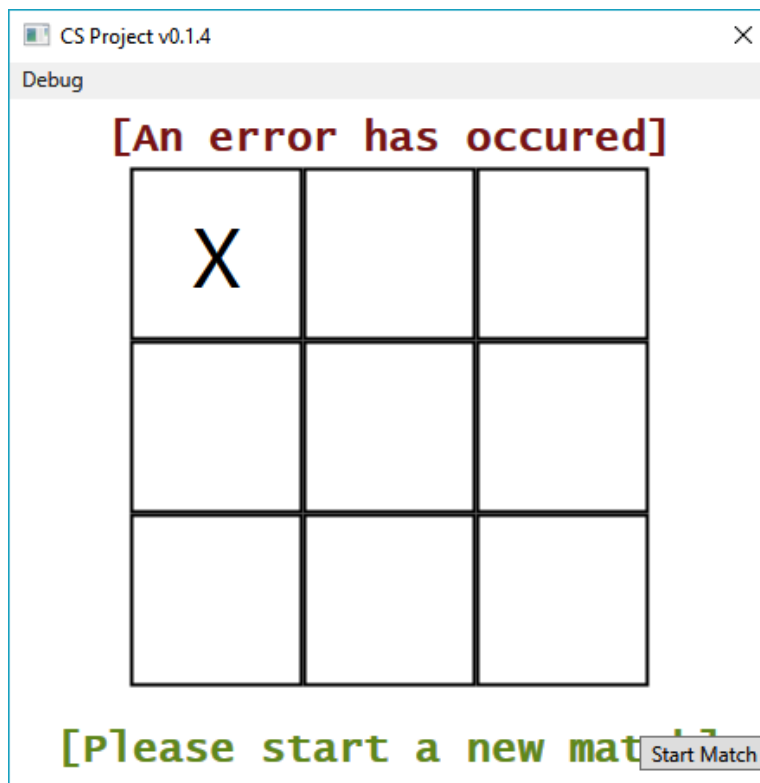


Passed Tests (10)	
✓ basicMatchTest	9 ms
✓ hashSerialiseTest	30 ms
✓ mergeTest	3 ms
✓ nodeSerialiseTest	4 ms
✓ removeTreeTest	27 ms
✓ saveTreeLoadTreeTest	56 ms
✓ statisticallyBestTest	5 ms
✓ testHash	50 ms
✓ treeExistsTest	5 ms
✓ walkTest	1 ms

[The new debug menu]



[The GUI after an error is caught. Shrunk slightly so it fits on the same page.]



I made the decision to disable the 'doWinBlockCheck' mode because first of all, it made winning pretty difficult (while its behaviour was much more human-like, I found myself tying too much for my liking).

After disabling the code that blocks the player from winning (but not the code for making the AI see a 'I can win with this move' opportunity), the AI felt much better to fight against; however, I felt that the 'doStatisticallyBest' mode (the main focus of the project) was just put out of focus. It was simply there to help guide the AI on a path where 'doWinBlockCheck' would make the AI win, which just doesn't 'feel' quite right, considering 'doStatisticallyBest' is what the AI should be relying on.

V0.3.0/V1.0.0

This is the final version of the game before it is deemed to be stable and completed. It mostly involves polishing what already exists, as well as adding tiny features (in other words, just picking off the low-hanging fruit).

Goals:

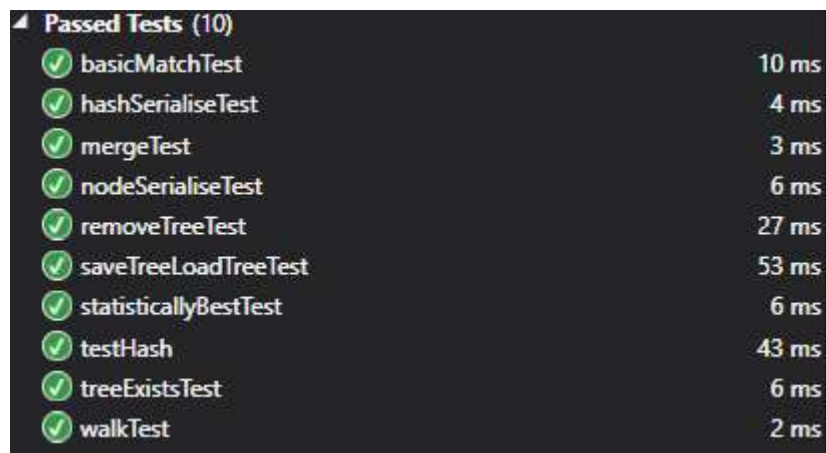
- Make the GameFiles.loadTree function throw an exception if the file has a version number it doesn't know about. **[V0.2.2]**
- ~~When the AI attempts to load the global tree, make a try-catch block around it, and if an exception is thrown while loading the tree, rename the file to something else then inform the user about it (by throwing another exception, so the error box shows up). This does mean that the AI loses all of its data (kinda, the file still exists though) but it also prevents the game from suddenly becoming unplayable. [Cancel that, I forgot how my own code worked ~.~, basically, the AI passes a flag to GameFiles.loadTree so it doesn't throw but simply returns null. This goal has already been met. Although... it made me aware that the flag never worked in the first place (time for more tests)]~~
- When the player's turn is over, the PlayerController should update the UI to say something like "The AI is thinking", so it feels a bit more natural when the AI is taking a while to figure out the next move. **[V0.2.3]**
- ~~Add an item to the GUI's menu that allows the user to select a file, and that file is then merged into their global tree. Call it something like 'Load External Data', and give it a tooltip explaining its function. This is useful, because it allows the user to import someone else's global tree, giving their own AI more data to work with. [This doesn't seem to be a terribly useful idea though, so I'm not going to implement it.]~~
- Currently, the MainWindow doesn't provide any functions for the PlayerController to modify it, the controller has to manually edit the GUI itself. Instead, provide functions such as "MainWindow.modifyText", "MainWindow.onGameStart", "MainWindow.onGameEnd", etc. so the controller has less knowledge about the GUI, and so MainWindow knows more about the state of the game. This will allow me, for example, to only let the user use the 'Load External Data' button if no game is running, but there is currently no reliable way for the MainWindow to know if a game is running without a function such as 'MainWindow.onGameEnd'. **[V0.2.1]**
- When the game is first loaded, display a message box detailing what tic-tac-toe is, and how to play it. Add a 'help' button to the menu so this box can be brought back up. **[V0.2.2]**
- More tests! I'm not terribly sure how well it would work, but I could fabricate calls to things like the function that is called when a grid button is clicked, and see if the results are correct.
- Make sure that every non-trivial function has a documentation comment. **[V0.2.3]**
- Modify the ISerialisable interface so that it takes a version number. **[V0.2.2]**
- Make Nodes and Hashes use a new, compressed format to store their data inside a file. **[V0.2.2]**
- Once MainWindow has been refactored to provide new functions to use, provide a way to lock/unlock the player from performing moves, so PlayerGUIController doesn't have to clear the message queue each time. **[V0.2.1]**
- Remove the placeholder text in the GUI, so things like "[User piece here]" doesn't display when the game is opened. **[V0.2.3]**

V0.2.1

Achieved:

- Create 'MainWindow.updateText' which is a function used to update the top and bottom text shown on screen. This means code no longer has to manually reference the labels.
- Create 'MainWindow.onEndMatch' which is called by the PlayerGUIController whenever a match ends, or by the game loop if an exception is thrown during a match.
- Add MainWindow.Flags, which is an enum of flags used to keep track of certain things.
- Add a mechanism into MainWindow which prevents the creation of PlayerPlaceMessages unless the MainWindow.Flags.CanPlacePiece flag is set. The only way to set this flag is through the MainWindow.unlockBoard function, which the PlayerGUIController now does.

Tests:



Passed Tests (10)	
✓ basicMatchTest	10 ms
✓ hashSerialiseTest	4 ms
✓ mergeTest	3 ms
✓ nodeSerialiseTest	6 ms
✓ removeTreeTest	27 ms
✓ saveTreeLoadTreeTest	53 ms
✓ statisticallyBestTest	6 ms
✓ testHash	43 ms
✓ treeExistsTest	6 ms
✓ walkTest	2 ms

Screenshot: [Nothing worth noting]

This version focuses completely on 'fortifying' the MainWindow class. This is accomplished by providing functions to do certain things (such as updating text on screen) so code no longer has to manually reference the controls, as well as making it stricter on how the player can influence the program (although, this is not noticeable to the player).

MainWindow also contains a bit more validation code to ensure it is more correct.

It should be noted, that at first, the game loop would call onEndMatch once a match finishes successfully, but this introduced a very noticeable delay from 'ending match' to 'start button reappears', making the UI feel sluggish, so I put the onEndMatch call into PlayerGUIController's code to make it happen faster.

With the change that introduced 'MainWindow.unlockBoard', it now means the PlayerGUIController no longer has to clean the message queue, while still allowing the user to only select a single move. This behaviour feels more 'correct' to me, than having it clean the message queue just as a workaround to a bug.

V0.2.2

Achieved:

- Throw an exception in GameFiles.loadTree if the tree's file version is higher than supported.
- Add the 'version' parameter to ISerialisable.deserialise, so the class knows the format of the input data.
- Bump the TREE file format to version 2.
- Hash.serialise and Hash.deserialise now use their TREE version 2 format.
- Node.serialise and Node.deserialise now use their TREE version 2 format.
- Add a help box that the game will show when it's ran for the first time. (This was supposed to be in version 0.2.3, but I forgot to bump the version up)

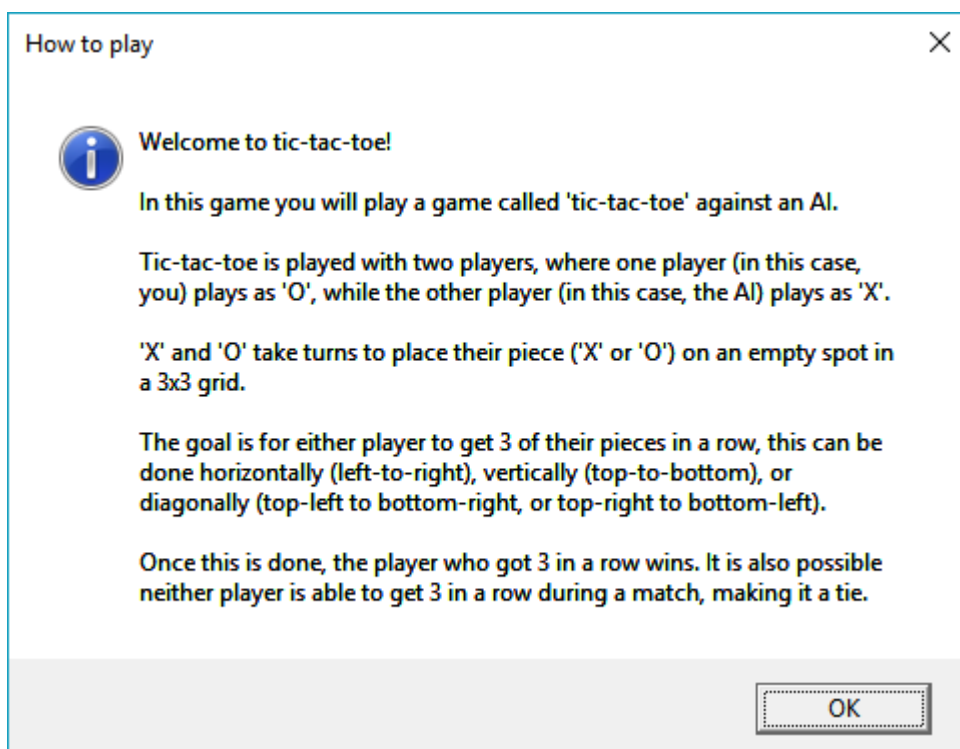
Tests:



Passed Tests (10)		
✓	basicMatchTest	8 ms
✓	hashSerialiseTest	4 ms
✓	mergeTest	3 ms
✓	nodeSerialiseTest	5 ms
✓	removeTreeTest	68 ms
✓	saveTreeLoadTreeTest	111 ms
✓	statisticallyBestTest	5 ms
✓	testHash	111 ms
✓	treeExistsTest	4 ms
✓	walkTest	1 ms

Screenshot:

[The new help box]



This update was mainly focused on making the game handle serialising its data more efficiently.

An interesting thing to note, is that when the exception was added to `GameFiles.loadTree`, it made me aware during testing that the code that constructs the AI wasn't protected by a try-catch statement which meant that the game simply crashed instead of displaying an error message. This was fixed, of course.

The new TREE version 2 format for the `Board.Hash` class now results in only **3** bytes per serialised hash, whereas TREE version 1 resulted in **12** bytes per hash. This is achieved by using as many bits as possible for the serialised data, and results in a **75%** reduction in size.

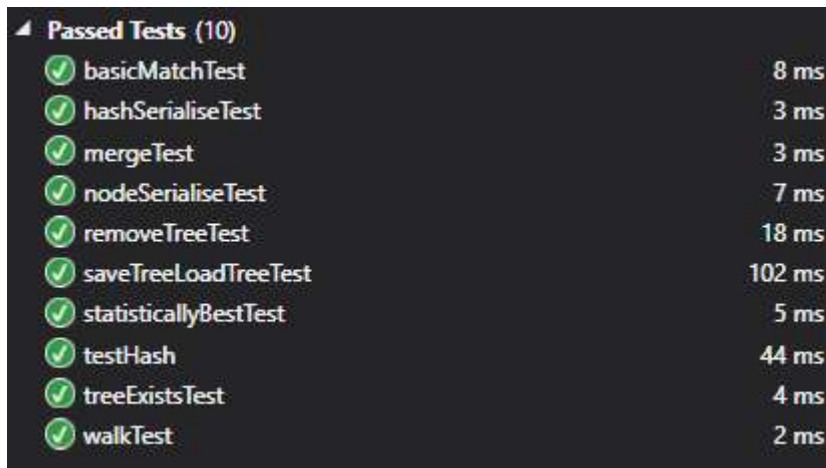
Less impressively, the TREE version 2 format for the `Node` class goes from **13** bytes (excluding the hash it serialises) to **10** bytes. Roughly a **23%** reduction.

V0.2.3

Achieved:

- Add an item onto the menu that, when clicked, displays the help box.
- Fix a bug where, if the user selected a non-empty slot, then the player controller wouldn't use `MainWindow.unlockBoard`, preventing the user from placing a piece, making the game 'soft-lock'. This bug was introduced with the new mechanism introduced in 0.2.1 for placing pieces on a board.
- Remove the placeholder text from the GUI.
- Change some on-screen text to show "It is your turn" when it's the player's turn, and "The AI is thinking..." when it's the AI's turn.
- Improve documentation and code style of all the code in the project.

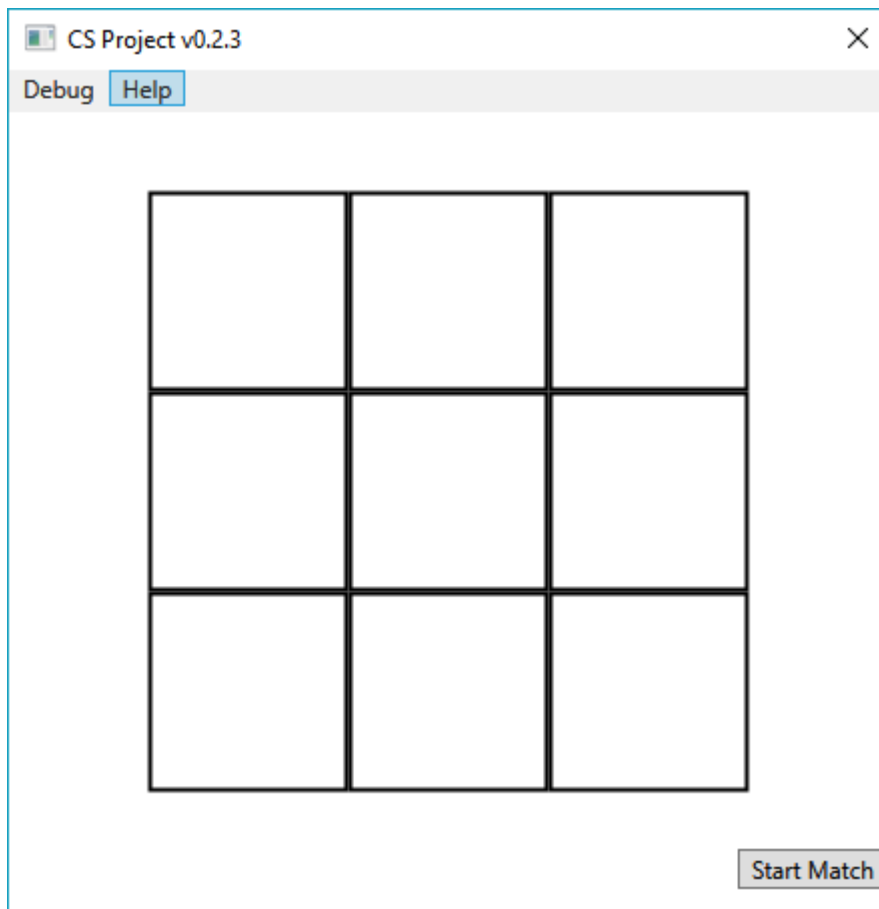
Tests:



Passed Tests (10)	
✓ basicMatchTest	8 ms
✓ hashSerialiseTest	3 ms
✓ mergeTest	3 ms
✓ nodeSerialiseTest	7 ms
✓ removeTreeTest	18 ms
✓ saveTreeLoadTreeTest	102 ms
✓ statisticallyBestTest	5 ms
✓ testHash	44 ms
✓ treeExistsTest	4 ms
✓ walkTest	2 ms

Screenshot: [Next page]

[The new 'Help' button. Also the placeholder text is now gone]



This version was just small leftovers that needed to be done before the project is deemed complete.

Chronological Order of Development

This table is a compilation of every development point from every version of the game and is placed in chronological order. This is here as evidence of the game's iterative development.

All dates are taken from when the changes were added to the git repository.

It should also be noted that a few points not mentioned earlier in the document will be present in this table.

Description	Version	Date
Have a way to hash the game board	0.0.1	5/12/2016 11:56 PM
Have a way to represent a tree of moves	0.0.1	15/12/2016 8:41 AM
Have a way to calculate the statistically best move to make	0.0.1	15/12/2016 10:10 AM
Include this document	0.0.1	15/12/2016 10:36 AM
Add basic serialisation support	0.0.2	26/12/2016 11:50 PM
Implement serialisation for the Hash class	0.0.2	26/12/2016 11:50 PM
Remove the MoveTree class, and move its functionality into the Node class	0.0.2	27/12/2016 00:03 AM
Implement serialisation for the Node class	0.0.2	27/12/2016 00:39 AM
Add the Board and Controller classes	0.0.2	28/12/2016 00:11 AM
Add the game logic thread, as well as a way for the UI and game threads to talk to each other	0.0.3	28/12/2016 1:31 AM
Add the GUI, as well as the PlayerController class	0.0.3	28/12/2016 4:21 AM
Fixed a bug in the win checking code, where a wrong slot was being checked	0.0.3	28/12/2016 5:43 PM
Prevent the player from placing a piece in a non-empty slot	0.0.3	28/12/2016 5:58 PM
Modify the Board class to support when two controllers tie	0.0.3	28/12/2016 7:01 PM
When the user hovers their mouse over a slot in the GUI's grid, change its background colour. Aka, make it more interactive.	0.0.3	29/12/2016 3:48 PM
Add a label to the GUI, informing the user what piece they're playing as	0.0.3	29/12/2016 3:49 PM
Put the Hash class under the Board namespace	0.0.4 0.1.0	04/01/2017 11:49 PM
Display the current version of the project in the window's title	0.1.1	07/01/2017 00:52 AM
Modify Node.walk to allow an Action<Node> to be passed to it	0.1.1	07/01/2017 1:58 AM
Create the function Node.walkEveryPath	0.1.1	07/01/2017 3:28 AM
Add a Debug window to easily view a tree of Nodes	0.1.2	10/01/2017 4:58 PM
Create static variable Config.versionString	0.1.2	10/01/2017 5:04 PM
Modify the Debug window so a piece of text on screen can be changed, so things like the AI can give some extra feedback	0.1.2	11/01/2017 6:24 AM
Add the first version of the AI. It only uses its 'doRandom' algorithm, and has an untested, and unused 'doStatisticallyBest' algorithm. (While the global move tree existed, there currently wasn't any data going into it. This mean that the AI always had to fall back onto doRandom)	0.1.2	11/01/2017 7:36 AM
Add a button to start a new match (also remove the requirement of having to restart the game to play more than one match)	0.1.2	11/01/2017 7:46 AM

Description	Version	Date
Add code to the AI so it merges its local move tree into the global one. (After this was done, I then manually tested the AI's doStatisticallyBest algorithm... and it worked without issue)	0.1.2	11/01/2017 8:51 AM
Modify the Controller base class, so certain functions are given more information	0.1.2	12/01/2017 9:16 AM
Add the MainWindow.reportException function, which will display an exception inside of a message box so the user knows something's gone wrong.	0.1.3	13/01/2017 6:55 AM
Add the GameFiles class	0.1.3	13/01/2017 7:52 AM
Modify the AI so it saves/loads its global tree to a file. (This surprisingly worked without any issue as well)	0.1.3	13/01/2017 8:05 AM
Add the Node.merge function	0.1.3	17/01/2017 00:13 PM
Improve Node.merge so the source tree (which goes into a destination tree) can have more than one path. (If there were more than one path, it simply wouldn't be merged over. The debug windows allowed me to catch this)	0.1.3	17/01/2017 1:21 PM
Begin the writeup of the project's report (I did do this severely out of order, I admit)	0.1.4	24/01/2017 3:11 PM
Add this table of chronological development	0.1.4	24/01/2017 11:08 PM
Fixed a bug where the GUI wouldn't update the 3x3 grid if the last piece placed was by the AI	0.1.4	26/01/2017 10:14 PM
Format the list of bugs for v0.2.0 as a table, instead of a list of bullet points	0.1.4	28/01/2017 2:44 AM
Fixed a bug where, if the player spam clicks empty slots on the 3x3 grid, it will queue up all of the moves and perform them one at a time	0.1.4	29/01/2017 3:09 AM
Add a debug menu to the GUI. It comes with a debug function to throw an exception in the game thread	0.1.4	29/01/2017 3:31 AM
Fixed a bug where, if the game thread threw an exception, then the game thread would be terminated; making the game unresponsive	0.1.4	29/01/2017 3:47 AM
Fixed a bug where an error box is shown anytime the game is closed.	0.1.4	29/01/2017 3:51 AM
Add Board.predict	0.1.4	31/01/2017 3:00 AM
Allow a controller to use Board.predict, but using the other controller's piece. (So the 'X' controller can use 'O' controller's piece with Board.predict)	0.1.4	31/01/2017 3:46 AM
Add the 'doWinBlockCheck' mode for the AI, albeit disabled.	0.1.4	31/01/2017 3:56 AM
Remove the code for 'doWinBlockCheck'. I can use a simple git command to get the code back. I just don't want it using up space.	0.1.4	31/01/2017 4:00 AM
Add a 25% chance to perform 'doRandom', if the AI's current path has less than 25% chance of winning.	0.1.4	31/01/2017 4:32 AM
Prevent the Node Debug windows from opening when DEBUG isn't defined.	0.1.4	31/01/2017 4:42 AM
Add MainWindow.updateText, as a way to update the text on the screen.	0.2.1	16/02/2017 2:10 PM
Add MainWindow.onEndMatch, and MainWindow.Flags	0.2.1	16/02/2017 2:35 PM

Description	Version	Date
Add MainWindow.unlockBoard, and MainWindow.Flags.CanPlacePiece	0.2.1	16/02/2017 3:05 PM
Throw exception in GameFiles.loadTree if the file version is not supported	0.2.2	16/02/2017 4:10 PM
Add the 'version' parameter to ISerialisable.deserialise	0.2.2	16/02/2017 4:25 PM
Implement Board.Hash's 'serialise' and 'deserialise' functions for their TREE version 2 format.	0.2.2	16/02/2017 5:32 PM
Implement Node's 'serialise' and 'deserialise' functions for their TREE version 2 format.	0.2.2	16/02/2017 5:45 PM
Add a help box that the game displays when it is ran for the first time.	0.2.2	20/02/2017 4:24 PM
Add an item to the menu that displays the help box.	0.2.3	23/02/2017 11:33 PM
Remove the placeholder text from the GUI.	0.2.3	26/02/2017 11:39 PM
Fix a bug that prevented the user from placing their piece on the board.	0.2.3	26/02/2017 11:42 PM
Change some text to "The AI is thinking..."	0.2.3	28/02/2017 11:27 PM
Improve documentation and tests for the GameFiles class.	0.2.3	28/02/2017 11:39 PM
Improve documentation and code style of all the code in the project.	0.2.3	01/03/2017 00:01 AM to 1:11 AM
Bump the project to v1.0.0	1.0.0	10/03/2017 1:26 PM

Code Appendix

This appendix contains all of the code making up the project. Page breaks have been inserted where necessary, as an attempt to prevent functions from being broken up among different pages. Indentation has been slightly modified in areas, to make code lines that span multiple document lines easier to read.

Unfortunately, I could not find a clean way to include the code, so some of it is an unreadable mess. Due to Visual Studio's auto-indenting not being something I like, some places in the code will be inconsistent (e.g I prefer "if()" whereas Visual Studio prefers "if ()". Notice the space.) I also use camelCasing as opposed to the standard PascalCasing used in C#, so there are some naming inconsistencies.

Board.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.Linq;
using CS_Project.Game.Controllers;

/// <summary>
/// Contains everything related to the game board.
/// </summary>
namespace CS_Project.Game
{
    /// <summary>
    /// Contains the state of the game board, and provides an interface to
    /// manipulate it.
    ///
    /// This class also provides a 'game loop' for a tic-tac-toe match.
    /// </summary>
    public partial class Board
    {
        /// <summary>
        /// The amount of pieces used on the board.
        /// </summary>
        public const uint pieceCount = 3*3;

        private const uint _xIndex = 0; // Index in _board for the x player
        private const uint _oIndex = 1; // Index in _board for the o player

        private Board.Piece[] _board { get; set; } // Current state of
the Board.
        private Board.Flags _flags { get; set; } // A set of
bitflags used to keep track of some stuff.
        private Board.Stage _stage { get; set; } // The current
state of the match.
        private Controller _current { get; set; } // The Controller
who currently has control of the board.
        private int _lastIndex { get; set; } // The last index
used to place a piece.
```

```
[Flags]
private enum Flags : byte
{
    HasSetPiece = 1 << 0 // Flag for whether the current controller
has set its piece. Used to check for correctness
}

private enum Stage
{
    Initialisation,
    InControllerTurn,    // The board is waiting for a controller
to perform it's turn.
    AfterControllerTurn, // The controller has just made its turn.
    NoMatch
}

/// <summary>
/// Creates a hash of the board, using a given piece as the "myPiece".
/// </summary>
///
/// <param name="piece">The piece that should be used as the
"myPiece"</param>
private Hash createHashFor(Board.Piece piece)
{
    var hash = new Hash(piece);

    for(var i = 0; i < this._board.Length; i++)
        hash.setPiece(this._board[i], i);

    return hash;
}
```

```

    /// <summary>
    /// Determines if anyone has won yet.
    /// </summary>
    ///
    /// <param name="isTie">Set to 'true' if there was a tie.</param>
    ///
    /// <returns>The piece that won, or Piece.empty if no one has won yet.</returns>
    private Piece checkForWin(out bool isTie)
    {
        // A closure that checks 3 spaces on the board, and returns
        true if they all are the 'p' piece.
        Func<uint, uint, uint, Piece, bool> check = null;
        check = delegate(uint i1, uint i2, uint i3, Piece p)
        {
            return this._board[i1] == p
                && this._board[i2] == p
                && this._board[i3] == p;
        };

        // Default isTie to false.
        isTie = false;

        // For both X and O, check all the possible win positions.
        var pieces = new Piece[]{ Board.Piece.X, Board.Piece.O };
        foreach(var piece in pieces)
        {
            if(check(0, 1, 2, piece)) return piece; // Top row
            if(check(3, 4, 5, piece)) return piece; // Middle row
            if(check(6, 7, 8, piece)) return piece; // Bottom row
            if(check(0, 4, 8, piece)) return piece; // Top left to
            bottom right, and vice-versa
            if(check(2, 4, 6, piece)) return piece; // Top right to
            bottom left, and vice-versa
            if(check(0, 3, 6, piece)) return piece; // Top left to
            bottom left, and vice-versa
            if(check(1, 4, 7, piece)) return piece; // Top middle to
            bottom middle, and vice-versa
            if(check(2, 5, 8, piece)) return piece; // Top right to
            bottom right, and vice-versa
        }

        // If there are no empty spaces, and the above checks didn't
        make the function return, then we've tied.
        var emptyCount = this._board.Count(p => p == Piece.Empty);
        isTie = (emptyCount == 0);

        return Piece.Empty;
    }

    /// <summary>
    /// Default constructor for a board.
    /// </summary>
    public Board()
    {
        this._stage = Stage.NoMatch;
        this._board = new Board.Piece[Board.pieceCount];

        for(var i = 0; i < this._board.Length; i++)
            this._board[i] = Piece.Empty;
    }

```

```

    /// <summary>
    /// Starts a match between two controllers.
    ///
    /// Note to self: Run all of this stuff in a seperate thread,
otherwise the GUI will freeze.
    /// Use System.Collections.Concurrent.ConcurrentQueue to talk between
the two threads.
    /// </summary>
    ///
    /// <param name="xCon">The controller for the X piece.</param>
    /// <param name="oCon">The controller for the O piece.</param>
    public void startMatch(Controller xCon, Controller oCon)
    {
        Debug.Assert(this._stage == Stage.NoMatch, "Attempted to start
a match while another match is in progress.");

        #region Setup controllers.
        Debug.Assert(xCon != null, "The X controller is null.");
        Debug.Assert(oCon != null, "The O controller is null.");
        this._stage = Stage.Initialisation;

        // Inform the controllers what piece they're using.
        xCon.onMatchStart(this, Piece.X);
        oCon.onMatchStart(this, Piece.O);

        // Reset some stuff
        this._lastIndex = int.MaxValue;
        #endregion

        #region Match turn logic
        Board.Piece turnPiece = Piece.O;    // The piece of who's turn
it is.
        Board.Piece wonPiece = Piece.Empty; // The piece of who's won.
Empty for no win.
        bool isTie = false;
        while (wonPiece == Piece.Empty && !isTie) // While there hasn't
been a tie, and no one has won yet.
        {
            // Unset some flags
            this._flags &= ~Flags.HasSetPiece;

            #region Do controller turn
            this._stage = Stage.InControllerTurn;
            var hash = this.createHashFor(turnPiece);
// Create a hash from the point of view of who's turn it is.
            var controller = (turnPiece == Piece.X) ? xCon : oCon;
// Figure out which controller to use this turn.
            this._current = controller;

            controller.onDoTurn(hash, this._lastIndex); // Allow the
controller to perform its turn.
            Debug.Assert((this._flags & Flags.HasSetPiece) != 0,
                $"The controller using the {turnPiece}
piece didn't place a piece.");
            #endregion

            #region Do after controller turn
            this._stage = Stage.AfterControllerTurn;
            hash = this.createHashFor(turnPiece); // Create
another hash for the controller

```

```

        controller.onAfterTurn(hash, this._lastIndex); // And let
the controller handle its 'after move' logic
        #endregion

        #region Misc stuff
        wonPiece = this.checkForWin(out isTie);
// See if someone's won/tied yet.
        turnPiece = (turnPiece == Piece.X) ? Piece.O : Piece.X;
// Change who's turn it is
        #endregion
    }
    #endregion
    Debug.Assert(wonPiece != Piece.Empty || isTie, "There was no
win condition, but the loop still ended.");

    #region Process the win
    // Create a hash for both controllers, then tell them whether
they tied, won, or lost.
    var stateX = this.createHashFor(Piece.X);
    var stateO = this.createHashFor(Piece.O);
    if(isTie)
    {
        xCon.onMatchEnd(stateX, this._lastIndex,
MatchResult.Tied);
        oCon.onMatchEnd(stateO, this._lastIndex,
MatchResult.Tied);
    }
    else if (wonPiece == Piece.O)
    {
        xCon.onMatchEnd(stateX, this._lastIndex,
MatchResult.Lost);
        oCon.onMatchEnd(stateO, this._lastIndex,
MatchResult.Won);
    }
    else
    {
        xCon.onMatchEnd(stateX, this._lastIndex,
MatchResult.Won);
        oCon.onMatchEnd(stateO, this._lastIndex,
MatchResult.Lost);
    }
    #endregion

    #region Reset variables
    this._stage = Stage.NoMatch;
    this._current = null;

    for (var i = 0; i < this._board.Length; i++)
        this._board[i] = Piece.Empty;
    #endregion
}

```

```

    /// <summary>
    /// Sets a piece on the board.
    /// </summary>
    ///
    /// <param name="index">The index of where to place the
piece.</param>
    /// <param name="controller">The controller that's placing the
piece.</param>
    public void set(int index, Controller controller)
    {
        // There are so many ways to use this function wrong...
        // But I need these checks here to make sure my code is correct.
        Debug.Assert(this._stage == Stage.InControllerTurn,
            "A controller attempted to place its piece outside of its
onDoTurn function.");

        Debug.Assert(this.isCurrentController(controller),
            "Something's gone wrong somewhere. An incorrect controller is
being used.");

        Debug.Assert(index < Board.pieceCount && index >= 0,
            $"Please use Board.pieceCount to properly limit the index.
Index = {index}");

        Debug.Assert((this._flags & Flags.HasSetPiece) == 0,
            "A controller has attempted to place its piece twice. This is a
bug.");

        Debug.Assert(this._board[index] == Piece.Empty,
            "A controller attempted to place its piece over another piece.
Enough information is passed to prevent this.");

        this._board[index] = controller.piece;
        this._lastIndex    = index;
        this._flags        |= Flags.HasSetPiece;
    }

    /// <summary>
    /// Determines if the given controller is the controller who's turn
it currently is.
    /// </summary>
    ///
    /// <param name="controller">The controller to check.</param>
    ///
    /// <returns>'true' if 'controller' is the controller's who's
controlling the current turn.</returns>
    public bool isCurrentController(Controller controller)
    {
        return (this._current == controller);
    }
}

```

```

// This part of 'Board' is used for anything that should be accessed like
"Board.Piece"
// Whereas the other part is for the actual board class.
// This is done simply because I prefer "Board.Piece.X" to "Piece.X"
public partial class Board
{

    /// <summary>
    /// Describes the different board pieces
    /// </summary>
    public enum Piece : byte
    {
        /// <summary>
        /// The X piece
        /// </summary>
        X = 0,

        /// <summary>
        /// The O piece
        /// </summary>
        O = 1,

        /// <summary>
        /// An empty board piece
        /// </summary>
        Empty = 2
    }

    /// <summary>
    /// Contains a hash of the game board.
    /// </summary>
    public class Hash : ICloneable, ISerialiseable
    {
        private char[] _hash; // The hash itself

        /// <summary>
        /// Char that represents the other player's piece
        /// </summary>
        public const char otherChar = 'O';

        /// <summary>
        /// Char that represents the piece of the player using this class
        /// </summary>
        public const char myChar = 'M';

        /// <summary>
        /// Char that represents an empty space
        /// </summary>
        public const char emptyChar = '.';

        /// <summary>
        /// The piece that the other player is using.
        /// </summary>
        public Board.Piece otherPiece { private set; get; }

        /// <summary>
        /// The piece that the user of this class is using.
        /// </summary>
        public Board.Piece myPiece { private set; get; }
    }
}

```



```

private Hash(Board.Piece myPiece, bool dummyParam)
{
    if (myPiece == Board.Piece.Empty)
        throw new HashException("myPiece must not be
Board.Piece.empty");

    // Figure out who is using what piece.
    this.myPiece = myPiece;
    this.otherPiece = (myPiece == Board.Piece.O) ? Board.Piece.X

: Board.Piece.O;
}

/// <summary>
/// Default constructor for a Hash
///
/// Note that the 'myPiece' for the hash will be 'Board.Piece.x'
/// </summary>
public Hash() : this(Board.Piece.X)
{ }

/// <summary>
/// Constructs a new Hash.
/// </summary>
///
/// <exception cref="CS_Project.Game.HashException">If `myPiece` is
`Board.Piece.empty`</exception>
///
/// <param name="myPiece">The piece that you are using, this is
needed so the class knows how to correctly format the hash.</param>
public Hash(Board.Piece myPiece) : this(myPiece, new
string(Hash.emptyChar, 9))
{
}

```

```

    /// <summary>
    /// Constructs a new Hash from a given hash string.
    /// </summary>
    ///
    /// <exception cref="CS_Project.Game.HashException">If `myPiece` is
    `Board.Piece.empty`</exception>
    ///
    /// <param name="myPiece">The piece that you are using, this is
    needed so the class knows how to correctly format the hash.</param>
    /// <param name="hash">
    ///     The hash string to use.
    ///
    ///     An internal check is made with every function call, that
    determines if the hash is still correct:
    ///     * The hash's length must be the same as
    'Board.pieceCount'
    ///     * The hash's characters must only be made up of
    'Hash.myChar', 'Hash.otherChar', and 'Hash.emptyChar'.
    ///
    ///     If the given hash fails to meet any of these checks, then an
    error box will be displayed.
    ///     In the future, when I can be bothered, exceptions will be
    thrown instead so the errors can actually be handled.
    /// </param>
    public Hash(Board.Piece myPiece, IEnumerable<char> hash) :
    this(myPiece, false)
    {
        this._hash = hash.ToArray();
        this.checkCorrectness();
    }

    /// <returns>The actual hash itself, properly formatted.</returns>
    public override string ToString()
    {
        this.checkCorrectness();
        return new string(this._hash); // Create an immutable copy of
the hash.
    }

```

```

    /// <summary>
    /// Sets a piece in the hash.
    /// </summary>
    ///
    /// <exception cref="System.ArgumentOutOfRangeException">If index is
    >= `Board.pieceCount`</exception>
    /// <exception cref="CS_Project.Game.HashException">If
    `allowOverwrite` is false, and there is a non-empty piece at
    'index'</exception>
    ///
    /// <param name="piece">The piece to use</param>
    /// <param name="index">The index to place the piece</param>
    /// <param name="allowOverwrite">See the `HashException` part of this
    documentation</param>
    public void setPiece(Board.Piece piece, int index, bool
    allowOverwrite = false)
    {
        // Enforce the behaviour of `allowOverwrite`
        if (this.getPieceChar(index) != Hash.emptyChar
        && !allowOverwrite)
            throw new HashException($"Attempted to place {piece} at
            index {index}, however a non-null piece is there and allowOverwrite is
            false. Hash = {this._hash}");

        // Figure out which character to use to represent `piece`.
        char pieceChar = '\0';

        if (piece == this.myPiece)    pieceChar = Hash.myChar;
        else if (piece == this.otherPiece) pieceChar = Hash.otherChar;
        else                          pieceChar = Hash.emptyChar;

        // Then place that character into the hash.
        this._hash[index] = pieceChar;
        this.checkCorrectness();
    }

```

```

    /// <summary>
    /// Gets the board piece at a certain index.
    /// </summary>
    ///
    /// <exception cref="System.ArgumentOutOfRangeException">If index is
>= `Board.pieceCount`</exception>
    ///
    /// <param name="index">The index to use</param>
    ///
    /// <returns>The board piece at 'index'</returns>
    public Board.Piece getPiece(int index)
    {
        Board.Piece piece = Board.Piece.Empty;
        var pieceChar      = this.getPieceChar(index);

        // Convert the character into a Board.Piece
        switch (pieceChar)
        {
            case Hash.emptyChar: piece = Board.Piece.Empty; break;
            case Hash.myChar:    piece = this.myPiece;      break;
            case Hash.otherChar: piece = this.otherPiece;   break;

            default: Debug.Assert(false, "This should not have
happened"); break;
        }

        return piece;
    }

    /// <summary>
    /// Determines if a specific piece in the hash is the user's.
    /// </summary>
    ///
    /// <exception cref="System.ArgumentOutOfRangeException">If index is
>= `Board.pieceCount`</exception>
    ///
    /// <param name="index">The index to check.</param>
    ///
    /// <returns>`true` if the piece at `index` belongs to the user of
this class. `false` otherwise.</returns>
    public bool isMyPiece(int index)
    {
        return this.getPieceChar(index) == Hash.myChar;
    }

    /// <summary>
    /// Determines if a specific piece in the hash is empty.
    /// </summary>
    ///
    /// <exception cref="System.ArgumentOutOfRangeException">If index is
>= `Board.pieceCount`</exception>
    ///
    /// <param name="index">The index to check.</param>
    ///
    /// <returns>`true` if the piece at `index` is empty. `false`
otherwise.</returns>
    public bool isEmpty(int index)
    {
        return this.getPieceChar(index) == Hash.emptyChar;
    }

```

```

    /// <summary>
    /// Clones the Hash.
    /// </summary>
    ///
    /// <returns>A clone of this instance of Hash.</returns>
    public object Clone()
    {
        return new Hash(this.myPiece, this.ToString());
    }

    /// <summary>
    /// Gets the character at a certain index in the hash.
    /// </summary>
    private char getPieceChar(int index)
    {
        this.enforceIndex(index);
        this.checkCorrectness();
        return this._hash[index];
    }

    /// <summary>
    /// In D, there's something called an 'invariant' function, which
    allows me to run code to make sure the class is still
    /// in correct condition after every function call.
    ///
    /// This doesn't seem to exist in C#, so this function is called by
    every other function to make sure it's still correct.
    /// (From Future Me: There does seem to be an invariant in C#, but
    it's far too late to bother changing this)
    ///
    /// This allows me to be confident that the class is working as it
    should be.
    /// </summary>
    private void checkCorrectness()
    {
        Debug.Assert(this._hash.Length == Board.pieceCount,
            $"The length of the hash is {this._hash.Length} when it should be {Board.pieceCount}");

        Debug.Assert(this._hash.All(c => (c == Hash.emptyChar || c ==
            Hash.myChar || c == Hash.otherChar)),
            $"The hash contains an invalid character.
Hash = {this._hash}");
    }

    /// <summary>
    /// Enforces that `index` is between 0 and Board.pieceCount
    /// </summary>
    ///
    /// <exception cref="System.ArgumentOutOfRangeException">Thrown if
    `index` is >= to `Board.pieceCount`</exception>
    ///
    /// <param name="index">The index to check</param>
    private void enforceIndex(int index)
    {
        if (index >= Board.pieceCount)
            throw new ArgumentOutOfRangeException("index", $"index
must be between 0 and {index} (exclusive)");
    }

```

```

// override object.Equals(auto generated with tweaks)
public override bool Equals(object obj)
{
    if (obj == null || GetType() != obj.GetType())
    {
        return false;
    }

    var other = (Hash)obj;
    return (other.ToString() == this.ToString()) && (other.myPiece
== this.myPiece);
}

// override object.GetHashCode(auto generated)
public override int GetHashCode()
{
    return base.GetHashCode();
}

// implement ISerialiseable.serialise
public void serialise(BinaryWriter output)
{
    this.checkCorrectness();
    Debug.Assert(this._hash.Length <= byte.MaxValue, "The hash's
length can't fit into a byte.");

    // See 'TREE version 2' in the deserialise function for the
format.

    byte[] bytes = new byte[3];
    for(int i = 0; i < Board.pieceCount; i++)
    {
        var piece      = this.getPiece(i); // The piece at the
slot.
        var byteIndex = (i * 2) / 8;      // Index into 'bytes'
for which byte to modify.
        var bitOffset = (i * 2) % 8;      // The offset into the
byte to write the data to.
        var pieceBits = 0;

        if(piece == Piece.Empty)    pieceBits = 0;
        else if(piece == this.myPiece) pieceBits = 1;
        else if(piece == this.otherPiece) pieceBits = 2;

        bytes[byteIndex] |= (byte)((byte)pieceBits << bitOffset);
    }

    // Put in the 'M' and 'O' bits
    // 0000 0100 = 0x4 (O = X, M = O)
    // 0000 1000 = 0x8 (O = O, M = X)
    bytes[2] |= (this.myPiece == Piece.O) ? (byte)0x4 : (byte)0x8;

    output.Write(bytes);
}

```

```

// implement ISerialiseable.deserialise
public void deserialise(BinaryReader input, uint version)
{
    // TREE version 1
    if(version == 1)
    {
        var length      = input.ReadByte();
        this._hash       = input.ReadChars(length);
        this.myPiece     = (Board.Piece)input.ReadByte();
        this.otherPiece  = (Board.Piece)input.ReadByte();
    }

    /**
     * Format of a hash: (TREE version 2)
     * [3 bytes]
     * byte 1: 4433 2211
     * byte 2: 8877 6655
     * byte 3: 0000 MO99
     *
     * Numbers such as '11' and '55' represent the Board.Piece in
slot '1' and '5', respectively.
     * 'M' and 'O' represent the Board.Piece of 'My' piece and
'Other' piece, respectively.
     * '0' Represents 'unused'
     *
     * For 'M' and 'O':
     * 0 = Board.Piece.O
     * 1 = Board.Piece.X
     *
     * So if the 'MO' bits were 0x40: M = O, O = X
     * If 'MO' were 0x80:           M = X, O = O
     *
     * For '11' to '99':
     * 0 = Empty
     * 1 = M
     * 2 = O
     * **/
    if (version == 2)
    {
        var bytes          = input.ReadBytes(3);
        var identityBits = bytes[2] & 0xC; // Identity = The bits
defining 'myPiece' and 'otherPiece'. 0xC = 1100
        this.myPiece      = (identityBits & 0x4) == 0x4 ?
Piece.O : Piece.X;
        this.otherPiece   = (identityBits & 0x4) == 0x4 ?
Piece.X : Piece.O;

        for (int i = 0; i < Board.pieceCount; i++)
        {
            var byteIndex = (i * 2) / 8; // Index into 'bytes'
for which byte to use.
            var bitOffset = (i * 2) % 8; // The offset into
the byte to write the data to.
            var byte_     = bytes[byteIndex];
            var piece     = (byte_ >> bitOffset) & 0x3; // 0x3
== 0000 0011

```

```

        switch(piece)
        {
            case 0: this._hash[i] = Hash.emptyChar; break;
            case 1: this._hash[i] = Hash.myChar; break;
            case 2: this._hash[i] = Hash.otherChar; break;

            default:
                throw new IOException("");
        }
    }

    this.checkCorrectness();
}
}

/// <summary>
/// (Auto generated from the 'Exception' code-snippet.)
///
/// Thrown whenever the 'Hash' class is used incorrectly.
/// </summary>
[Serializable]
public class HashException : Exception
{
    public HashException() { }
    public HashException(string message) : base(message) { }
    public HashException(string message, Exception inner) : base(message,
inner) { }
    protected HashException(
        System.Runtime.Serialization.SerializationInfo info,
        System.Runtime.Serialization.StreamingContext context) : base(info,
context) { }
}
}

```


Config.cs

```

namespace CS_Project.Game
{
    /// <summary>
    /// Contains some static/constant data used to change a few things.
    /// </summary>
    public static class Config
    {
        /// <summary>
        /// A string form of the project's current version.
        /// </summary>
        public const string versionString = "v1.0.0";

        /// <summary>
        /// The text to be displayed in the game's help box.
        ///
        /// While this would make more sense to be put in the 'MainWindow'
        class, it'd also look a bit ugly (since it's so large).
        /// </summary>
        public const string helpBoxInfo
        = "Welcome to tic-tac-toe!\n\n"

        + "In this game you will play a game called 'tic-tac-toe' against an
        AI.\n\n"

        + "Tic-tac-toe is played with two players, where one player (in this case,
        you) plays as 'O', "
        + "while the other player (in this case, the AI) plays as 'X'.\n\n"

        + "'X' and 'O' take turns to place their piece ('X' or 'O') on an empty
        spot in a 3x3 grid.\n\n"

        + "The goal is for either player to get 3 of their pieces in a row, this
        can be done "
        + "horizontally (left-to-right), vertically (top-to-bottom), or diagonally
        (top-left to bottom-right, or top-right to bottom-left).\n\n"

        + "Once this is done, the player who got 3 in a row wins. "
        + "It is also possible neither player is able to get 3 in a row during a
        match, making it a tie.";
    }
}

```

GameFiles.cs

```

using System;
using System.IO;

namespace CS_Project.Game
{
    /// <summary>
    /// A static class containing functions related to storing/reading files
    /// associated with the game.
    /// </summary>
    public static class GameFiles
    {
        // Data paths
        private const string _dataFolder = "data/";
        // Path to where all the data is stored.
        private const string _treeFolder = _dataFolder + "trees/";
        // Path to where the trees are stored.

        private const string _helpFileFlag = _dataFolder + "help_was_shown"; //
        // Path to where the "show help" file is stored.

        // File format info (For .tree files)
        public const byte treeFileVersion = 2; // This is the
        // version that the class supports. It can read older versions, but not newer
        private const string _treeFileHeader = "TREE"; // The "magic number"
        // for a .tree file.

        /// <summary>
        /// Combines `path` with the path to the tree folder.
        /// </summary>
        private static string makeTreePath(string path)
        {
            var finalPath = Path.Combine(GameFiles._treeFolder, path +
            ".tree");

            return finalPath;
        }

        /// <summary>
        /// Makes sure the data folder, and the tree folder exist.
        /// </summary>
        private static void ensureDirectories()
        {
            // The directories to check for.
            var directories = new string[] { GameFiles._dataFolder,
            GameFiles._treeFolder };

            foreach(var dir in directories)
            {
                if(!Directory.Exists(dir))
                    Directory.CreateDirectory(dir);
            }
        }
    }
}

```

```
/// <summary>
/// Determines whether a tree with a specific name exists.
/// </summary>
///
/// <param name="name">The name of the tree to search for.</param>
///
/// <returns>True if the tree exists. False otherwise.</returns>
public static bool treeExists(string name)
{
    return File.Exists(GameFiles.makeTreePath(name));
}

/// <summary>
/// Removes a tree with a specific name.
/// </summary>
///
/// <exception cref="System.IO.FileNotFoundException">Thrown if
`shouldThrow` is true, and a tree called `name` does not exist.</exception>
///
/// <param name="name">The name of the tree to remove</param>
/// <param name="shouldThrow">If True, then this function will throw
a FileError if the tree does not exist.</param>
public static void removeTree(string name, bool shouldThrow = false)
{
    if(shouldThrow && !GameFiles.treeExists(name))
        throw new FileNotFoundException("Could not remove tree as
it does not exist.", GameFiles.makeTreePath(name));

    File.Delete(GameFiles.makeTreePath(name));
}
```

```

    /// <summary>
    /// Saves a tree to a file.
    /// </summary>
    ///
    /// <exception cref="System.ArgumentNullException">Thrown if `root`
is null.</exception>
    /// <exception cref="System.IO.IOException">Thrown if `overwrite` is
false, and a tree called `name` already exists.</exception>
    ///
    /// <param name="name">The name to give the tree.</param>
    /// <param name="root">The root node of the tree.</param>
    /// <param name="overwrite">
    ///     If True, then if a tree called 'name' already exists, it is
overwritten.
    ///     If False, then an IOException is thrown if a tree called
'name' already exists.
    /// </param>
public static void saveTree(string name, Node root, bool overwrite = true)
{
    if(root == null)
        throw new ArgumentNullException("root");

    if(!overwrite && GameFiles.treeExists(name))
        throw new IOException($"Unable to save tree {name} as it
already exists, and overwrite is set to false.");

    // Make sure the tree directory exists, then serialise `root`
into a new tree file.
    GameFiles.ensureDirectories();
    using (var fs = new FileStream(GameFiles.makeTreePath(name),
    FileMode.Create))
    {
        using (var bw = new BinaryWriter(fs))
        {
            /*
            * Format:
            * [4 bytes, 'TREE']
            * [1 byte, file version]
            * [X bytes, serialised tree]
            * */

            bw.Write((char[])GameFiles._treeFileHeader.ToCharArray());
            bw.Write((byte)GameFiles.treeFileVersion);
            root.serialise(bw);
        }
    }

    /// <summary>
    /// Loads a previously saved tree.
    /// </summary>
    ///
    /// <exception cref="System.IO.FileNotFoundException">Thrown if a
tree called `name` doesn't exist.</exception>
    /// <exception cref="System.IO.IOException">Thrown if the tree file
is malformed in some way, and cannot be loaded.</exception>
    ///
    /// <param name="name">The name of the tree to load.</param>
    /// <param name="shouldThrow">If True, then an exception is thrown if
'name' doesn't exist.</param>

```

```

    ///
    /// <returns>
    ///     The root node of the tree saved as 'name'.
    ///     Null is returned if 'shouldThrow' is False, and 'name'
doesn't exist.
    /// </returns>
    public static Node loadTree(string name, bool shouldThrow = true)
    {
        try
        {
            if(!GameFiles.treeExists(name))
                throw new FileNotFoundException($"Unable to load
tree {name} as it does not exist.", GameFiles.makeTreePath(name));

            using (var fs = new
FileStream(GameFiles.makeTreePath(name), FileMode.Open))
            {
                using (var br = new BinaryReader(fs))
                {
                    // Make sure the header is correct
                    string header = new
string(br.ReadChars(GameFiles._treeFileHeader.Length));
                    if(header != GameFiles._treeFileHeader)
                        throw new IOException($"Unable to load
tree {name} as it's header is incorrect: '{header}'");

                    // Then read in the version number, and make
sure we can read it in.
                    byte version = br.ReadByte();
                    if(version > GameFiles.treeFileVersion)
                        throw new IOException($"Unable to load
tree {name} as it uses a newer version of the TREE format.\n"
                                                + $"File
version: {version} | Highest Supported version:
{GameFiles.treeFileVersion}");

                    // Then unserialise the tree.
                    var root = Node.root;
                    root.deserialise(br, version);

                    return root;
                }
            }
        }
        catch(Exception ex)
        {
            if(shouldThrow)
                throw ex;

            return null;
        }
    }
}

```

```
    /// <summary>
    /// This function determines if the game should display the help
message box.
    ///
    /// If this function returns `true`, a file is created in the 'data'
folder which will then
    /// make this function always return `false` while the file exists.
    /// </summary>
    /// <returns>`true` if the help box should be displayed. `false`
otherwise.</returns>
    public static bool shouldShowHelpMessage()
    {
        if(!File.Exists(GameFiles._helpFileFlag))
        {
            File.Create(GameFiles._helpFileFlag).Dispose();
            return true;
        }

        return false;
    }
}
```

ISerialisable.cs

```

using System.IO;

namespace CS_Project.Game
{
    /// <summary>
    /// An interface that any class that can be serialised/unserialised should
    /// implement.
    ///
    /// The reason a custom serialiser interface is used instead of
    /// using .Net's serialisation stuff, is because
    /// I want my code to have readonly, private, private set, etc. variables,
    /// but that doesn't play well with
    /// how .Net's serialisation seems to work.
    /// </summary>
    interface ISerialiseable
    {
        /// <summary>
        /// Writes data into the binary writer that can later be used to
        /// deserialise the object.
        /// </summary>
        ///
        /// <param name="output">The output stream to write to</param>
        void serialise(BinaryWriter output);

        /// <summary>
        /// Reads data from the binary writer and changes the object to
        /// reflect the deserialised data.
        /// </summary>
        ///
        /// <param name="input">The input stream to read from.</param>
        /// <param name="version">A version number used to specify the format
        /// of the data in `input`.</param>
        void deserialise(BinaryReader input, uint version);
    }
}

```

Message.cs

```

using CS_Project.Game.Controllers;

namespace CS_Project.Game
{
    /// <summary>
    /// The base class for any class that can be sent over the message queue.
    /// </summary>
    public abstract class Message
    {
    }

    /// <summary>
    /// The message sent when the game thread should startup a match.
    /// </summary>
    public sealed class StartMatchMessage : Message
    {
        /// <summary>
        /// The controller for the 'X' piece.
        /// </summary>
        public Controller xCon { get; set; }

        /// <summary>
        /// The controller for the 'Y' piece.
        /// </summary>
        public Controller oCon { get; set; }
    }

    /// <summary>
    /// The message that is sent whenever the player chooses where to place his
    piece.
    /// </summary>
    public sealed class PlayerPlaceMessage : Message
    {
        /// <summary>
        /// The index of the slot that the player wants to place their slot in.
        /// </summary>
        public int index { get; set; }
    }

    /// <summary>
    /// This test message is used to tell the game thread to throw an Exception.
    /// </summary>
    public sealed class ThrowExceptionMessage : Message
    {
    }
}

```


Node.cs

```

using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;

namespace CS_Project.Game
{
    // This is a private class used in Node.merge to keep track of some data.
    // NodeMergeInfo keeps track of two versions of a node, and an index.
    // The 'node' Node is the version of the node that's in the 'source' tree.
    // The 'parent' Node is the version of the node that's in the 'destination'
    tree
    // The index is used as an index to 'node.children'.
    class NodeMergeInfo
    {
        internal Node node    { set; get; } // Node being used. (This node is
        in the 'source' tree)
        internal Node parent { set; get; } // The version of 'node' that is
        inside the 'destination' tree.
        internal int index   { set; get; } // Index used for node.children
    }

    /// <summary>
    /// A node describing a single move of a tic-tac-toe match.
    ///
    /// A node may also contain other children nodes, which together represent
    a tree of moves.
    /// </summary>
    public class Node : ICloneable, ISerialiseable
    {
        /// <summary>
        /// The hash of the board after the move was made.
        /// </summary>
        public Board.Hash hash { private set; get; }

        /// <summary>
        /// The index of what slot was changed this move.
        /// </summary>
        public uint index { private set; get; }

        /// <summary>
        /// How many times this move was used in a game that was won.
        /// </summary>
        public uint won;

        /// <summary>
        /// How many times this move was used in a game that was lost.
        /// </summary>
        public uint lost;

        /// <summary>
        /// This node's children.
        /// </summary>
        public List<Node> children { set; get; }
    }
}

```

```

    /// <summary>
    /// Calculates the percentage of games that have been won.
    /// </summary>
    public float winPercent
    {
        get
        {
            float total = (this.won + this.lost);
            return (this.won / total) * 100.0f;
        }
    }

    /// <summary>
    /// Calculates the percentage of games that have been lost.
    /// </summary>
    public float losePercent
    {
        get
        {
            float total = (this.won + this.lost);
            return (this.lost / total) * 100.0f;
        }
    }

    /// <summary>
    /// Creates a node suitable for use as a root node.
    /// </summary>
    public static Node root
    {
        get
        {
            return new Node(new Board.Hash(Board.Piece.X,
"....."), uint.MaxValue);
        }
    }

    /// <summary>
    /// Creates a new Node
    /// </summary>
    ///
    /// <exception cref="System.ArgumentNullException">When `hash` is
null.</exception>
    ///
    /// <param name="hash">The 'Board.Hash' of the board after the move
was made.</param>
    /// <param name="index">The index of the slot that was
changed.</param>
    /// <param name="won">How many times this move was used in a won
match.</param>
    /// <param name="lost">The opposite of 'won'.</param>
    public Node(Board.Hash hash, uint index, uint won = 0, uint lost = 0)
    {
        if (hash == null)
            throw new ArgumentNullException("hash");

        this.hash      = hash;
        this.index      = index;
        this.won        = won;
        this.lost       = lost;
        this.children   = new List<Node>();
    }

```

```
/// <summary>
/// Default constructor for a Node.
/// </summary>
public Node()
{
    this.children = new List<Node>();
    this.hash     = new Board.Hash();
}

/// <summary>
/// Clones the node, and all of it's children.
/// </summary>
///
/// <returns>A clone of this node.</returns>
public object Clone()
{
    var toReturn = new Node((Board.Hash) this.hash.Clone(),
this.index, this.won, this.lost);

    foreach(var child in this.children)
        toReturn.children.Add((Node)child.Clone());

    return toReturn;
}
```

```

    /// <summary>
    /// Given a list of hashes, walks through this tree of nodes, up to a
certain depth, and performs
    /// an action on every node walked to.
    /// </summary>
    ///
    /// <exception cref="System.ArgumentNullException">Thrown if `path`
or `action` is null.</exception>
    /// <exception cref="System.ArgumentOutOfRangeException">Thrown if
`depth` is 0.</exception>
    ///
    /// <param name="path">The path of hashes to follow.</param>
    /// <param name="action">The action to perform on every node
followed.</param>
    /// <param name="depth">The maximum amount of nodes to walk to
(inclusive).</param>
    ///
    /// <returns>
    ///     'true' if the entire 'path' was walked, or if 'depth' amount
of nodes were walked to.
    ///     'false' if the 'path' was cut short.
    /// </returns>
    public bool walk(ICollection<Board.Hash> path, Action<Node> action, uint
depth = uint.MaxValue)
    {
        if (path == null)
            throw new ArgumentNullException("path");

        if (action == null)
            throw new ArgumentNullException("action");

        if (depth == 0)
            throw new ArgumentOutOfRangeException("depth", "The depth
must be 1 or more");

        uint walked = 0;
        var currentThis = this; // Current node in this
tree
        var currentPath = path.GetEnumerator(); // Current hash in the
path
        currentPath.MoveNext();

        // While:
        //     We haven't walked the full depth.
        // AND There are still nodes in the path we need to follow.
        while (walked < depth && walked < currentPath.Count)
        {
            walked += 1;

            bool found = false; // If 'true', then a node from the
path was found. If 'false', then the path ended prematurely
            foreach (var node in currentThis.children)
            {
                if (node.hash.Equals(currentPath.Current))
                {
                    currentPath.MoveNext();
                    currentThis = node;
                    found = true;

                    action(node);
                    break;
                }
            }
        }
    }

```

```
        }  
    }  
    if (!found)  
        return false;  
}  
return true;  
}
```

```

    /// <summary>
    /// Walks over every possible path in the node tree, and calls an
    action on each path.
    /// </summary>
    ///
    /// <exception cref="System.ArgumentNullException">Thrown if `action`
    is null.</exception>
    ///
    /// <param name="action">
    ///     The action to perform on every path.
    ///     Note that the parameter given is not a copy of the original,
    so anytime the parameter is stored somewhere,
    ///     a copy should be created.
    /// </param>
    public void walkEveryPath(Action<List<Node>> action)
    {
        if(action == null)
            throw new ArgumentNullException("action");

        var path = new List<Node>(); // Stores the nodes that make up
        the current path.

        // A closure, which is used recursively to travel through the
        tree.
        Func<Node, bool, bool> walk = null;
        walk = delegate (Node node, bool noAdd)
        {
            if(!noAdd) // Used so we don't add in the root node
                path.Add(node);

            // If the node is childless, then we're at the end of a
            path, so perform the action on it.
            if(node.children.Count == 0)
            {
                action(path);
            }
            else
            {
                // Otherwise, go through all of the node's children,
                and call 'walk' on them as well.
                foreach(var child in node.children)
                {
                    walk(child, false);
                    path.RemoveAt(path.Count - 1); // Shrink the
                    list by 1 once its done, so we don't have to create a new list.
                }
            }

            return false; // Dummy value
        };

        // Walk through this node first. This node is treated as the
        root, so is never added to any paths.
        walk(this, true);
    }

```

```

    /// <summary>
    /// Merges the nodes (including the wins/losses counter) from a
source tree, into a destination tree.
    ///
    /// Any nodes in 'source' that don't belong in 'destination' will be
created.
    /// </summary>
    ///
    /// <exception cref="System.ArgumentNullException">Thrown if
'destination' or 'source' is null.</exception>
    ///
    /// <param name="destination">The tree that will be modified.</param>
    /// <param name="source">The tree providing the nodes to
merge.</param>
    public static void merge(Node destination, Node source)
    {
        if(destination == null)
            throw new ArgumentNullException("destination");
        if(source == null)
            throw new ArgumentNullException("source");

        // If, for whatever reason, the source tree is empty. Then
return, otherwise we'll crash a few lines down.
        if (source.children.Count == 0)
            return;

        var info      = new Stack<NodeMergeInfo>();           // Too
annoying to explain, but this allows nodes with multiple children to be
merged.
        var parent    = destination;                         // Current
node in the destination tree. Starts off at the root.
        var local     = new NodeMergeInfo {node = source}; // Current
node in the source tree.
        local.parent = parent;
        while (true)
        {
            // Get the next local node.
            // local will be set to null if there are no nodes left.
            while(true)
            {
                // First, get how many children the node will have
left for us.
                var childCount = local.node.children.Count -
local.index;

                // Then, if there's still children, push the
current local onto the info stack, and make the next child the new local.
                if(childCount > 0)
                {
                    info.Push(local);
                    local = new NodeMergeInfo { node =
local.node.children[local.index++] };
                    break;
                }
                else
                {
                    // Otherwise, if there's no children left.
                    if(info.Count == 0) // Break if the stack is
empty
                {

```

```

        local = null;
        break;
    }

    // Otherwise, pop the stack, set it to local,
and run the loop over it.
    local = info.Pop();
    parent = local.parent;
}

// If there are no more nodes left.
if(local == null)
    break;

// Go over all the children in the parent.
Node node = null;
foreach (Node child in parent.children)
{
    // If the child is in the source tree, then set it
as 'node'
    if (child.hash.ToString() ==
local.node.hash.ToString())
    {
        node = child;
        break;
    }

    // If no matching node was found, create it!
    if (node == null)
    {
        // We create a new node so we don't add in all the
children with it
        // (while correct, it breaks my original
visualisation of the algorithm. I need this here to keep it sane in my
head)
        node = new Node(local.node.hash, local.node.index);
        parent.children.Add(node);
    }

    // Then add in the win/loss counters
    node.won += local.node.won;
    node.lost += local.node.lost;

    // Set the new parents
    parent = node;
    local.parent = parent;
}
}

```



```

// implement ISerialisable.serialize
public void serialize(BinaryWriter output)
{
    // See the deserialize function for the latest format.
    Debug.Assert(this.children.Count <= byte.MaxValue, "For some
reason, this Node has over 255 children e_e?");

    this.hash.serialize(output);
    output.Write((byte)this.index);
    output.Write((uint)this.won);
    output.Write((uint)this.lost);
    output.Write((byte)this.children.Count);

    foreach(var node in this.children)
        node.serialize(output);
}

// implemented ISerialisable.deserialize
public void deserialize(BinaryReader input, uint version)
{
    /*
    * Format of a serialised Node(TREE version 1 & 2):
    * [Serialised Board.Hash of the Node]
    * [4 bytes, Node.index] (In TREE version 2, this is a single byte)
    * [4 bytes, Node.won]
    * [4 bytes, Node.lost]
    * [1 byte, Node.children.count]
    * [All of the nodes children are then serialised.]
    */
    if(version == 1 || version == 2)
    {
        this.hash.deserialize(input, version);
        this.index = (version == 1) ? input.ReadUInt32() :
input.ReadByte();
        this.won = input.ReadUInt32();
        this.lost = input.ReadUInt32();

        var count = input.ReadByte();
        this.children = new List<Node>();
        for(int i = 0; i < count; i++)
        {
            var node = new Node();
            node.deserialize(input, version);
            this.children.Add(node);
        }
    }
}

```

```

/// <summary>
/// Contains a node path (array of nodes), and can calculate the average
win percentage of the path.
///
/// Also contains static functions to help find, for example, the path in a
tree that will most likely result in a win.
/// </summary>
public class Average
{
    /// <summary>
    /// The node path
    /// </summary>
    public List<Node> path;

    /// <summary>
    /// Calculates the average win percentage of the path.
    ///
    /// This is an O(n) algorithm, where 'n' is how many nodes are in the
    `path`.
    /// </summary>
    public float averageWinPercent
    {
        get
        {
            if(this.path.Count == 0) // Avoid divide-by-zero errors
                return 0;

            float totalPercent = 0;
            this.path.ForEach(node => totalPercent +=
node.winPercent);

            return (totalPercent / this.path.Count);
        }
    }

    /// <summary>
    /// Creates a new Average.
    /// </summary>
    public Average()
    {
        this.path = new List<Node>();
    }
}

```

```

    /// <summary>
    /// Finds the path in 'root' that is statistically the most likely to
win.
    /// </summary>
    ///
    /// <exception cref="System.ArgumentNullException">Thrown if `root`
is null.</exception>
    ///
    /// <param name="root">The root node to search through.</param>
    ///
    /// <returns>An 'Average' containing the path with the highest win
percent.</returns>
    public static Average statisticallyBest(Node root)
    {
        if (root == null)
            throw new ArgumentNullException("root");

        // All this function does, is keep track of the path with the
highest averageWinPercent.
        // This shows off the power/reusability of Node.walkEveryPath
        var pathAverage = new Average();
        var bestAverage = new Average();

        root.walkEveryPath(path =>
        {
            pathAverage.path = path;

            if (pathAverage.averageWinPercent >
bestAverage.averageWinPercent)
                bestAverage.path = new
List<Node>(pathAverage.path); // Copies the list
            });
        return bestAverage;
    }
}

```

Controllers/Controller.cs

```

namespace CS_Project.Game.Controllers
{
    /// <summary>
    /// Contains the result of a match.
    /// </summary>
    public enum MatchResult
    {
        /// <summary>
        /// The controller won the match.
        /// </summary>
        Won,

        /// <summary>
        /// The controller lost the match.
        /// </summary>
        Lost,

        /// <summary>
        /// The controller tied with the other one.
        /// </summary>
        Tied
    }

    /// <summary>
    /// The base class for any class that can control the game board.
    /// </summary>
    public abstract class Controller
    {
        /// <summary>
        /// The instance of 'Board' that is currently using this controller.
        /// </summary>
        public Board board { private set; get; }

        /// <summary>
        /// The Board.Piece that this controller has been assigned to for the
        match.
        /// </summary>
        public Board.Piece piece { private set; get; }

        /// <summary>
        /// Called whenever a new match is started.
        /// </summary>
        ///
        /// <param name="board">The board that is using this
        controller.</param>
        /// <param name="myPiece">Which piece this controller has been
        given.</param>
        public virtual void onMatchStart(Board board, Board.Piece myPiece)
        {
            this.board = board;
            this.piece = myPiece;
        }
    }
}

```

```

    /// <summary>
    /// Called whenever the match has ended.
    ///
    /// Notes for inheriting classes: Call 'super.onMatchEnd' only at the
end of the function.
    /// </summary>
    ///
    /// <param name="boardState">The final state of the board.</param>
    /// <param name="index">The index of where the last piece was placed
on the board.</param>
    /// <param name="result">Contains the match result.</param>
    public virtual void onMatchEnd(Board.Hash boardState, int index,
MatchResult result)
    {
        this.board = null;
        this.piece = Board.Piece.Empty;
    }

    /// <summary>
    /// Called whenever the controller has to process its turn.
    /// </summary>
    ///
    /// <param name="boardState">
    ///     The hash of the current state of the board.
    ///
    ///     The hash's 'myPiece' is set to the same one given to the
controller with the 'onMatchStart' function.
    /// </param>
    /// <param name="index">
    ///     The index of the last move the other controller made.
    ///     If the other controller hasn't made a move yet, then this
will be int.MaxValue
    /// </param>
    public abstract void onDoTurn(Board.Hash boardState, int index);

    /// <summary>
    /// Called after the controller has taken its turn.
    /// </summary>
    ///
    /// <param name="boardState">The state of the board after the
controller's turn.</param>
    /// <param name="index">The index of where the last piece was placed
on the board.</param>
    public abstract void onAfterTurn(Board.Hash boardState, int index);
}
}

```

Controllers/AI.cs

```

using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;

namespace CS_Project.Game.Controllers
{
    /// <summary>
    /// The controller that provides the AI.
    /// </summary>
    public sealed class AI : Controller
    {
        // Trees
        private Node _globalTree;
        private Node _localTree;

        // Data needed for creating the localTree/performing a move.
        private bool _useRandom; // If true, then the AI will perform a
        random move. This is used as a fallback.
        private Random _rng;

        // Other
        private NodeDebugWindow _debug; // Debug window for general
        info + local tree
        private NodeDebugWindow _globalDebug; // Debug window purely for the
        global tree.
        private const string _globalName = "global_move_tree"; // The name
        used to save the global move tree.

        /// <summary>
        /// Performs an action using the debug window's dispatcher (only if
        it's not null).
        /// </summary>
        private void doDebugAction(Action action)
        {
            if (this._debug != null)
                this._debug.Dispatcher.Invoke(action);
        }
    }
}

```

```

    /// <summary>
    /// Adds a new node to the end of the local tree.
    /// </summary>
    ///
    /// <param name="hash">The hash of the board.</param>
    /// <param name="index">The index of where the piece was
placed.</param>
    private void addToLocal(Board.Hash hash, int index)
    {
        // This is a cheeky way to add onto the end of the local tree.
        // Since there is only a single path in the local tree, this is
fine.
        this._localTree.walkEveryPath(path =>
        {
            var node = new Node(hash, (uint)index);

            if (path.Count == 0)
                this._localTree.children.Add(node);
            else
                path.Last().children.Add(node);
        });

        // Update the local tree debugger. A clone is made in case the
tree is edited before the debug window finishes updating.
        this.doDebugAction(() =>
this._debug.updateNodeData((Node) this._localTree.Clone()));
    }

```

```

    /// <summary>
    /// Constructs a new version of the AI.
    ///
    /// The AI will attempt to load its global tree when constructed.
    /// </summary>
    ///
    /// <param name="window">A debug window for the local tree.</param>
    /// <param name="globalWindow">A debug window for the global
tree.</param>
    public AI(NodeDebugWindow window = null, NodeDebugWindow globalWindow
= null)
    {
        // Show the debug windows.
        if(window != null)
        {
            this._debug = window;
            window.Show();
        }

        if(globalWindow != null)
        {
            this._globalDebug = globalWindow;
            globalWindow.Show();
        }

        // Setup variables.
        this._useRandom = false;
        this._rng = new Random();

        // Load the move tree
        this._globalTree = GameFiles.loadTree(AI._globalName, false);
        if(this._globalTree == null)
            this._globalTree = Node.root;
    }

    // implement Controller.onMatchStart
    public override void onMatchStart(Board board, Board.Piece myPiece)
    {
        base.onMatchStart(board, myPiece);

        // Reset some variables.
        this._localTree = Node.root;
        this._useRandom = false;

        // Update the global tree debugger
        this.doDebugAction(() =>
this._globalDebug.updateNodeData(this._globalTree));
        this.doDebugAction(() =>
this._globalDebug.updateStatusText("[GLOBAL MOVE TREE DEBUGGER]"));
    }

```



```

    // implement Controller.onMatchEnd
    public override void onMatchEnd(Board.Hash state, int index,
MatchResult result)
    {
        // The windows won't be closed, as I may still need them.
        // I can just close them manually afterwards.
        base.onMatchEnd(state, index, result);

// If the last piece placed was by the other controller, then it won't have
a node in the local tree.
        // So we quickly add it.
        if(!state.isMyPiece(index))
            this.addToLocal(state, index);

// Now, the amount of nodes in the local tree should be the same as:
Board.pieceCount - amountOfEmptySlots
// If not, then we've not created a node somewhere.
// (This test was created to prevent this bug from happening again. Praise
be for the debug windows.)
        var emptyCount = 0; // How many spaces are empty
        for(int i = 0; i < Board.pieceCount; i++) // Count the empty
spaces.

            emptyCount += (state.isEmpty(i)) ? 1 : 0;

        this._localTree.walkEveryPath(path =>
        {
            // Then make sure the tree's length is the same
            var amountOfMoves = Board.pieceCount - emptyCount;
            Debug.Assert(path.Count == amountOfMoves,
                $"We've haven't added enough nodes to the local tree!\n"
                + $"empty = {emptyCount} | amountOfMoves = {amountOfMoves}
| treeLength = {path.Count}");

            // Finally, bump the won/lost counters in the local tree
            foreach(var node in path)
            {
                if(result == MatchResult.Won)
                    node.won += 1;
                else if(result == MatchResult.Lost)
                    node.lost += 1;
                else
                {
                    // If we tie, don't bump anything up.
                }
            }
        });

// Then merge the local tree into the global one.
Node.merge(this._globalTree, this._localTree);

// Save the global tree, and update the debug window.
GameFiles.saveTree(AI._globalName, this._globalTree);
this._globalDebug.updateNodeData(this._globalTree);
    }

```

```
// implement Controller.onAfterTurn
public override void onAfterTurn(Board.Hash boardState, int index)
{
    // Add the AI's move.
    this.addToLocal(boardState, index);
}

// implement Controller.onDoTurn
public override void onDoTurn(Board.Hash boardState, int index)
{
    // Add the other controller's last move, if they made one.
    if(index != int.MaxValue)
        this.addToLocal(boardState, index);

    if(this._useRandom)
        this.doRandom(boardState);
    else
        this.doStatisticallyBest(boardState);
}
```

```

    // Uses the statisticallyBest method for choosing a move.
    private void doStatisticallyBest(Board.Hash hash)
    {
        this.doDebugAction(() => this._debug.updateStatusText("Function
doStatisticallyBest was chosen.));

        Node parent = null; // This is the node that will be used as
the root in statisticallyBest

        // If our local tree has some nodes in it, then...
        if(this._localTree.children.Count > 0)
        {
            // First, get the path of the local tree.
            List<Node> localPath = null;
            this._localTree.walkEveryPath(path => localPath = new
List<Node>(path));

            // Then, attempt to walk through the global tree, and find the last node in
the path.

            Node last = null;
            var couldWalk = this._globalTree.walk(localPath.Select(n
=> n.hash).ToList(), n => last = n);

            // If we get null, or couldn't walk the full path, then fallback to
doRandom

            if(!couldWalk || last == null)
            {
                this._useRandom = true;
                this.doRandom(hash);
                return;
            }

            parent = last;
        }
        else // Otherwise, the global tree's root is the parent.
            parent = this._globalTree;

        // Then use statisticallyBest on the parent, so we can figure out our next
move.

        var average = Average.statisticallyBest(parent);

        // If Average.statisticallyBest fails, fall back to doRandom.
        // Or, if the average win percent of the path is less than 25%, then
there's a 25% chance to do a random move.
        if(average.path.Count == 0
|| (average.averageWinPercent < 25.0 && this._rng.NextDouble() < 0.25))
        {
            this._useRandom = true;
            this.doRandom(hash);
            return;
        }

        // Otherwise, get the first node. Make sure it's a move we make.
        Then perform it!
        var node = average.path[0];
        Debug.Assert(node.hash.isMyPiece((int)node.index), "Something's
gone a *bit* wrong.");

        base.board.set((int)node.index, this);
    }

```

```
// Uses the randomAll method for choosing a move.
private void doRandom(Board.Hash hash)
{
    this.doDebugAction(() => this._debug.updateStatusText("Function
doRandom was chosen.));

    // Tis a bit naive, but meh.
    // Just keep generating a random number between 0 and 9
(exclusive) until we find an empty slot.
    while(true)
    {
        var index = this._rng.Next(0, (int)Board.pieceCount);

        if(hash.isEmpty(index))
        {
            this.board.set(index, this);
            break;
        }
    }
}
}
```

Controllers/PlayerGUI.cs

```

using System;
using System.Threading;

namespace CS_Project.Game.Controllers
{
    /// <summary>
    /// This controller is used alongside the GUI to allow the player to see,
    /// and interact with the game board.
    ///
    /// IMPORTANT SELF NOTE: Remember that this controller runs in the *Game*
    /// thread, not the *GUI* thread.
    /// </summary>
    class PlayerGUIController : Controller
    {
        private MainWindow _window { get; set; }

        /// <summary>
        /// Updates the GUI to reflect the new state of the board.
        /// </summary>
        ///
        /// <param name="boardState">The new state of the board.</param>
        /// <param name="turn">Who's turn it currently is.</param>
        private void updateGUI(Board.Hash boardState, Board.Piece turn)
        {
            this._window.Dispatcher.Invoke(() =>
            {
                this._window.updateBoard(boardState);
                this._window.updateText(null, (turn == base.piece) ? "It
is your turn" : "The AI is thinking...");
            });
        }

        /// <summary>
        /// Constructor for the controller.
        /// </summary>
        ///
        /// <exception cref="System.ArgumentNullException">Thrown if `window`
is null.</exception>
        ///
        /// <param name="window">The window that is displaying the
GUI.</param>
        public PlayerGUIController(MainWindow window)
        {
            if(window == null)
                throw new ArgumentNullException("window");

            this._window = window;
        }
    }
}

```

```

// implement Controller.onMatchStart
public override void onMatchStart(Board board, Board.Piece myPiece)
{
    base.onMatchStart(board, myPiece);

    // When the match starts, tell the player which piece they're using.
    this._window.Dispatcher.Invoke(() =>
    {
        this._window.updateText($"[You are {myPiece}]");
    });
}

// implement Controller.onMatchEnd
public override void onMatchEnd(Board.Hash state, int index,
MatchResult result)
{
    // Once the match has ended, figure out who won, and generate
the appropriate win message.
    string message = "";
    var enemyPiece = (base.piece == Board.Piece.X) ?
        Board.Piece.O : Board.Piece.X;

    if(result == MatchResult.Won) message = $"You
({base.piece}) have won!";
    else if(result == MatchResult.Lost) message = $"The enemy
({enemyPiece}) has won!";
    else if(result == MatchResult.Tied) message = "It's a tie! No
one wins.";
    else message = "[Unknown
result]";

    // Then update the GUI to display who's won.
    this.updateGUI(state, base.piece);
    this._window.Dispatcher.Invoke(() =>
    {
        this._window.updateText(null, message);
        this._window.onEndMatch();
    });

    base.onMatchEnd(state, index, result);
}

// implement Controller.onAfterTurn
public override void onAfterTurn(Board.Hash boardState, int index)
{
    // After the player has done their turn, update the GUI to
display it's the enemy's turn.
    this.updateGUI(boardState, (base.piece == Board.Piece.O) ?
Board.Piece.X : Board.Piece.O);
}

```

```

        // implement Controller.onDoTurn
        public override void onDoTurn(Board.Hash boardState, int index)
        {
            // Update the GUI to display the opponent's last move, as well
            // as to tell the user it's their turn.
            this.updateGUI(boardState, base.piece);
            // Let the player choose their piece
            // Note: This does not go through the dispatcher, since it can make it seem
            // like the GUI drops input
            // (due to the latency of Dispatcher.Invoke). It *shouldn't* create a data-
            // race, since nothing should be accessing it when this code is running.
            // It's worth keeping this line in mind though, future me, in case strange
            // things happen.
            this._window.unlockBoard();

            // Wait for the GUI to have signaled that the player has made a move.
            Message msg;
            while(true)
            {
                // Check every 50ms for a message.
                // If we didn't use a sleep, then the CPU usage skyrockets.
                if(!this._window.gameQueue.TryDequeue(out msg))
                {
                    Thread.Sleep(50);
                    continue;
                }

                // If we get a message not meant for us, requeue it.
                if(!(msg is PlayerPlaceMessage))
                {
                    this._window.gameQueue.Enqueue(msg);
                    continue;
                }

                // Otherwise, see if the placement is valid, and perform it.
                var info = msg as PlayerPlaceMessage;
                if(!boardState.isEmpty(info.index))
                {
                    this._window.unlockBoard(); // Unlock the board,
                    // otherwise the game soft-locks
                    continue;
                }

                this.board.set(info.index, this);
                break;
            }
        }
    }
}

```

NodeDebugWindow.xaml

```

<Window x:Class="CS_Project.Game.NodeDebugWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
    xmlns:local="clr-namespace:CS_Project.Game"
    mc:Ignorable="d"
    Title="NodeDebugWindow" Height="300" Width="300">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition/>
            <RowDefinition Height="21"/>
        </Grid.RowDefinitions>
        <ScrollViewer Grid.Row="0"
            HorizontalAlignment="Stretch"
            Height="Auto" VerticalAlignment="Stretch" Width="Auto"
            HorizontalScrollBarVisibility="Auto"
            VerticalScrollBarVisibility="Auto">

            <TreeView x:Name="tree" HorizontalAlignment="Stretch"
                Height="Auto" VerticalAlignment="Stretch" Width="Auto">
                <TreeViewItem x:Name="root" Header="Nodes"
                    IsExpanded="True" />
            </TreeView>
        </ScrollViewer>
        <TextBox x:Name="status" Grid.Row="1"
            HorizontalAlignment="Stretch" VerticalAlignment="Stretch"
            Width="Auto" Height="Auto"
            IsReadOnly="True"/>
    </Grid>
</Window>

```


MainWindow.xaml

```

<Window x:Class="CS_Project.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
        xmlns:local="clr-namespace:CS_Project"
        mc:Ignorable="d"
        Title="CS Project" Height="460" Width="455" ResizeMode="NoResize">
    <Window.Resources>
        <Style TargetType="Label"
                BasedOn="{StaticResource {x:Type Label}}"
                x:Name="SlotLabel">
            <Setter Property="FontSize" Value="48"/>

            <Style.Triggers>
                <Trigger Property="IsMouseOver" Value="True">
                    <Setter Property="Background" Value="#686868" />
                </Trigger>
            </Style.Triggers>
        </Style>
    </Window.Resources>

    <Grid>
        <Grid HorizontalAlignment="Center" Height="300" Margin="0,0,0,0"
VerticalAlignment="Center" Width="300">
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="100" />
                <ColumnDefinition Width="100" />
                <ColumnDefinition Width="100" />
            </Grid.ColumnDefinitions>
            <Grid.RowDefinitions>
                <RowDefinition Height="100" />
                <RowDefinition Height="100" />
                <RowDefinition Height="100" />
            </Grid.RowDefinitions>

            <Border BorderBrush="Black" BorderThickness="2" Grid.Column="0"
Grid.Row="0"/>
            <Border BorderBrush="Black" BorderThickness="2" Grid.Column="1"
Grid.Row="0"/>
            <Border BorderBrush="Black" BorderThickness="2" Grid.Column="2"
Grid.Row="0"/>
            <Border BorderBrush="Black" BorderThickness="2" Grid.Column="0"
Grid.Row="1"/>
            <Border BorderBrush="Black" BorderThickness="2" Grid.Column="1"
Grid.Row="1"/>
            <Border BorderBrush="Black" BorderThickness="2" Grid.Column="2"
Grid.Row="1"/>
            <Border BorderBrush="Black" BorderThickness="2" Grid.Column="0"
Grid.Row="2"/>
            <Border BorderBrush="Black" BorderThickness="2" Grid.Column="1"
Grid.Row="2"/>
            <Border BorderBrush="Black" BorderThickness="2" Grid.Column="2"
Grid.Row="2"/>

            <Label x:Name="slot0" Content="" HorizontalAlignment="Stretch"
Margin="0,0,0,0" VerticalAlignment="Stretch"

```

```

        Grid.Column="0" Grid.Row="0"
HorizontalContentAlignment="Center" VerticalContentAlignment="Center"/>
        <Label x:Name="slot1" Content="" HorizontalAlignment="Stretch"
Margin="0,0,0,0" VerticalAlignment="Stretch"
        Grid.Column="1" Grid.Row="0"
HorizontalContentAlignment="Center" VerticalContentAlignment="Center"/>
        <Label x:Name="slot2" Content="" HorizontalAlignment="Stretch"
Margin="0,0,0,0" VerticalAlignment="Stretch"
        Grid.Column="2" Grid.Row="0"
HorizontalContentAlignment="Center" VerticalContentAlignment="Center"/>
        <Label x:Name="slot3" Content="" HorizontalAlignment="Stretch"
Margin="0,0,0,0" VerticalAlignment="Stretch"
        Grid.Column="0" Grid.Row="1"
HorizontalContentAlignment="Center" VerticalContentAlignment="Center"/>
        <Label x:Name="slot4" Content="" HorizontalAlignment="Stretch"
Margin="0,0,0,0" VerticalAlignment="Stretch"
        Grid.Column="1" Grid.Row="1"
HorizontalContentAlignment="Center" VerticalContentAlignment="Center"/>
        <Label x:Name="slot5" Content="" HorizontalAlignment="Stretch"
Margin="0,0,0,0" VerticalAlignment="Stretch"
        Grid.Column="2" Grid.Row="1"
HorizontalContentAlignment="Center" VerticalContentAlignment="Center"/>
        <Label x:Name="slot6" Content="" HorizontalAlignment="Stretch"
Margin="0,0,0,0" VerticalAlignment="Stretch"
        Grid.Column="0" Grid.Row="2"
HorizontalContentAlignment="Center" VerticalContentAlignment="Center"/>
        <Label x:Name="slot7" Content="" HorizontalAlignment="Stretch"
Margin="0,0,0,0" VerticalAlignment="Stretch"
        Grid.Column="1" Grid.Row="2"
HorizontalContentAlignment="Center" VerticalContentAlignment="Center"/>
        <Label x:Name="slot8" Content="" HorizontalAlignment="Stretch"
Margin="0,0,0,0" VerticalAlignment="Stretch"
        Grid.Column="2" Grid.Row="2"
HorizontalContentAlignment="Center" VerticalContentAlignment="Center"/>
    </Grid>
    <Label x:Name="userPieceLabel" Content=""
        HorizontalAlignment="Stretch" Margin="0,27,0,0"
VerticalAlignment="Top" HorizontalContentAlignment="Center"
        FontSize="24" FontFamily="Lucida Console" FontWeight="Bold"
        Foreground="#FF781313" Style="{x:Null}"/>
    <Label x:Name="turnLabel" Content=""
        HorizontalAlignment="Stretch" Margin="0,380,0,0"
VerticalAlignment="Top" HorizontalContentAlignment="Center"
        FontSize="24" FontFamily="Lucida Console" FontWeight="Bold"
        Foreground="#FF5F871A" Style="{x:Null}" />
    <Button x:Name="startButton" Content="Start Match"
HorizontalAlignment="Left" Margin="364,389,0,0" VerticalAlignment="Top"
Width="75"
        Visibility="Visible" Click="onStartMatch"/>

    <DockPanel x:Name="toolBar" HorizontalAlignment="Stretch"
Margin="0,0,0,0" VerticalAlignment="Top">
        <Menu Height="21">
            <MenuItem Header="Debug" Visibility="{x:Static
local:MainWindow.debugVisibility}">
                <MenuItem x:Name="debug_throwException" Header="Throw
an Exception in game thread" Click="debug_throwException_Click" />
            </MenuItem>

            <MenuItem x:Name="menu_help" Header="Help"
Click="menu_help_Click"/>

```

```
        </Menu>  
    </DockPanel>  
</Grid>  
</Window>
```

NodeDebugWindow.xaml.cs

```
using System.Windows;
using System.Windows.Controls;

namespace CS_Project.Game
{
    /// <summary>
    /// Interaction logic for NodeDebugWindow.xaml
    /// </summary>
    public partial class NodeDebugWindow : Window
    {
        public NodeDebugWindow()
        {
            InitializeComponent();

            this.Title += $" {Config.versionString}";
        }
    }
}
```

```

/// <summary>
/// Updates the data shown in the tree view.
/// </summary>
///
/// <param name="root">The root of the tree to display.</param>
public void updateNodeData(Node root)
{
    this.root.Items.Clear();

    root.walkEveryPath(path =>
    {
        TreeViewItem parent = this.root;

        foreach(var node in path)
        {
            // First, make sure we're not duplicating nodes.
            // We do this by using the node's hash as the TreeViewItem's name.
            // So we can just check the names to make sure we're not duplicating data
            var nodeName = node.hash.ToString().Replace('.', '_');
            // WPF names can't use full stops.
            bool doContinue = false;
            foreach(ItemsControl con in parent.Items)
            {
                if(con.Name == nodeName)
                {
                    parent = con as TreeViewItem;
                    doContinue = true;
                    break;
                }
            }

            if(doContinue)
                continue;

            // Then, make the new node
            var item = new TreeViewItem();
            item.Header
            = "-----\n"
            + $"Hash: {node.hash}\n"
            + $"Index: {node.index}\n"
            + $"Wins: {node.won}({node.winPercent}%)\n"
            + $"Losses: {node.lost}({node.losePercent}%)\n";

            item.Name = nodeName;
            item.IsExpanded = true;

            // Finally, add it to the tree
            parent.Items.Add(item);
            parent = item;
        }
    });
}

```

```
    /// <summary>
    /// Updates the status text in the debug window. (The textbox at the
bottom)
    /// </summary>
    ///
    /// <param name="status">The status to change to.</param>
    public void updateStatusText(string status)
    {
        if(status == null)
            status = "";

        this.status.Text = status;
    }
}
```

MainWindow.xaml.cs

```

using System;
using System.Collections.Concurrent;
using System.Linq;
using System.Threading;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;

using CS_Project.Game;
using CS_Project.Game.Controllers;

namespace CS_Project
{
    /// <summary>
    /// Represents the state of the game thread.
    /// </summary>
    enum GameState
    {
        Waiting,    // The game thread is waiting for messages to be sent to
it.
        DoingMatch, // The game thread is processing a match.
        Crashed     // The game thread has thrown an exception.
    }

    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        /// <summary>
        /// Flags used for the `_flags` field, to keep track of some stuff.
        /// </summary>
        private enum Flags : byte
        {
            GameRunning    = 1 << 0, // Signals that a game is currently
running.
            CanPlacePiece  = 1 << 1, // Signals that the player is allowed
to try and place a piece.
        }

        private Thread _gameThread;
        private AI _aiInstance;    // Since the AI has
no way of reading in past data yet, I need to keep a single instance of it
so it can 'remember' past games.
        private Label[] _slots;    // The labels that
represent the slots on the board.
        private Flags _flags;      // See the individual
flags for details.
        public ConcurrentQueue<Message> gameQueue; // Used so the gui
thread can talk to the game thread.
    }
}

```

```

public MainWindow()
{
    // Setup multi-threaded stuff.
    InitializeComponent();
    this.gameQueue = new ConcurrentQueue<Message>();
    this._gameThread = new Thread(gameThreadMain);
    this._gameThread.Start();

    // Setup events
    base.Closed += MainWindow_Closed;

    // Setup the slots.
    this._slots = new Label[] { slot0, slot1, slot2,
                                slot3, slot4, slot5,
                                slot6, slot7, slot8 };
    foreach(var slot in this._slots)
        slot.MouseLeftButtonUp += onSlotPress;

    // Misc.
    this.Title += $" {Config.versionString}";

    if(GameFiles.shouldShowHelpMessage())
        this.showHelpBox();
}

// A simple function, that simply shows the help box.
private void showHelpBox()
{
    MessageBox.Show(Config.helpBoxInfo, "How to play",
        MessageBoxButton.OK, MessageBoxImage.Information);
}

// Abort the game thread once the window is closed.
// If we don't abort the game thread, then the program will stay
// alive in the background.
private void MainWindow_Closed(object sender, EventArgs e)
{
    this._gameThread.Abort();
}

```



```

    // When one of the slots are pressed, send a PlayerPlaceMessage to
the game thread, saying
    // which slot was pressed.
    private void onSlotPress(object sender, MouseEventArgs e)
    {
        // Don't do anything if the player isn't allowed to place a
piece yet.
        if ((this._flags & Flags.CanPlacePiece) == 0)
            return;

        // Lock the game board
        this._flags &= ~Flags.CanPlacePiece;

        // Only labels should be using this
        var label = sender as Label;

        // The last character of the labels is an index.
        var index = int.Parse(label.Name.Last().ToString());

        this.gameQueue.Enqueue(new PlayerPlaceMessage { index = index });
    }

    /// <summary>
    /// Updates the game board to reflect the given hash.
    /// </summary>
    ///
    /// <param name="hash">The 'Hash' containing the state of the
board.</param>
    public void updateBoard(Board.Hash hash)
    {
        // Figure out which characters to use.
        var myChar    = (hash.myPiece    == Board.Piece.X) ? "X" : "O";
        var otherChar = (hash.otherPiece == Board.Piece.X) ? "X" : "O";

        // Then fill out the game board.
        for(var i = 0; i < this._slots.Length; i++)
        {
            var slot = this._slots[i];

            if(hash.isMyPiece(i)) slot.Content = myChar;
            if(!hash.isMyPiece(i)) slot.Content = otherChar;
            if(hash.isEmpty(i))    slot.Content = "";
        }
    }
}

```

```

    /// <summary>
    /// Updates the two text labels on the screen.
    ///
    /// If a parameter is `null`, then the label won't be changed.
    /// </summary>
    ///
    /// <param name="topText">The text for the top of the screen.</param>
    /// <param name="bottomText">The text for the bottom of the
screen.</param>
    public void updateText(string topText, string bottomText = null)
    {
        if(topText != null)
            this.userPieceLabel.Content = topText;

        if(bottomText != null)
            this.turnLabel.Content = bottomText;
    }

    /// <summary>
    /// This function is called on the game thread to signal that the
match has ended.
    /// </summary>
    ///
    /// <exception cref="System.InvalidOperationException">Thrown if this
function is called when a match isn't in progress.</exception>
    public void onEndMatch()
    {
        if((this._flags & Flags.GameRunning) == 0)
            throw new InvalidOperationException("Attempted to call
onEndMatch while no game is running");

        // Unset some flags
        this._flags &= ~Flags.GameRunning;
        this._flags &= ~Flags.CanPlacePiece;

        // Make the start button visible again.
        this.startButton.Visibility = Visibility.Visible;
    }

```

```

/// <summary>
/// This function is called when the 'Start Match' button is pressed.
/// It begins a match between the AI and the player.
/// </summary>
private void onStartMatch(object sender, RoutedEventArgs e)
{
    // Error checking
    if((this._flags & Flags.GameRunning) > 0)
        throw new InvalidOperationException("Attempted to call
onStartMatch while a game is already running");

    // First, hide the button from being pressed again, and set
some flags.
    this.startButton.Visibility = Visibility.Hidden;
    this._flags |= Flags.GameRunning;
    this._flags |= Flags.CanPlacePiece;

    // Then, start up a match between the AI and the player
    if(this._aiInstance == null) // Use only a single instance of
the AI, so it doesn't have to reload the global tree over and over.
    {
        try
        {
            // In debug builds, give the AI debug windows.
            #if DEBUG
                this._aiInstance = new AI(new NodeDebugWindow(),
new NodeDebugWindow());
            #else
                this._aiInstance = new AI(null, null);
            #endif
        }
        catch(Exception ex)
        {
            this.reportException(ex);
            this.updateText("[THE AI CANNOT BE LOADED]",
"[ERROR]");

            this.onEndMatch();
            return;
        }
    }

    // Then send a message to start a match.
    this.gameQueue.Enqueue(new StartMatchMessage
    {
        xCon = new PlayerGUIController(this),
        oCon = this._aiInstance
    });
}

```

```

        /// <summary>
        /// Used by debug controls to control whether they're visible on
screen or not.
        ///
        /// If DEBUG is defined, the controls will be visible.
        /// Otherwise, the controls will not be visible.
        /// </summary>
        public static Visibility debugVisibility
        {
            get
            {
                #if DEBUG
                    return Visibility.Visible;
                #else
                    return Visibility.Collapsed;
                #endif
            }

            // Throws an exception in the game thread.
            // Used for testing reasons.
            private void debug_throwException_Click(object sender,
RoutedEventArgs e)
            {
                this.gameQueue.Enqueue(new ThrowExceptionMessage());
            }

            // Shows the help box when the user clicks 'Help'
            private void menu_help_Click(object sender, RoutedEventArgs e)
            {
                this.showHelpBox();
            }
        }

        // This part of the partial MainWindow class is for anything ran on or
related to the Game thread.
        public partial class MainWindow : Window
        {
            /// <summary>
            /// Called by the Game thread to signal that the player is allowed to
place their piece again.
            /// </summary>
            public void unlockBoard()
            {
                this._flags |= Flags.CanPlacePiece;
            }
        }

```

```
    // Any exception thrown in the game thread is passed to this function
(In the UI thread)
    // so that the UI can inform the user about something going wrong.
    public void reportException(Exception ex)
    {
        // Create the message to show the user.
        string msg = $"Something went wrong: {ex.Message}";

        // In debug mode, show the stack trace.
        #if DEBUG
            msg += @"\n{ex.StackTrace}";
        #endif

        MessageBox.Show(msg, "An exception was thrown",
        MessageBoxButton.OK, MessageBoxImage.Error);
    }
```

```

// The 'entry point' for the game thread.
private void gameThreadMain()
{
    // All variables for the game thread should be kept inside this
function.
    // Use Dispatcher.Invoke when the game thread needs to modify
the UI.
    // Use gameQueue so the GUI thread can speak to the game thread.
    var board = new Board();
    var state = GameState.Waiting;

    while (true)
    {
        try
        {
            // If no match is being done, listen to the message
queue for things.
            if(state == GameState.Waiting)
            {
                // Check for a message every 0.05 seconds.
                Message msg;
                while (!this.gameQueue.TryDequeue(out msg))
                    Thread.Sleep(50);

                // If we get a StartMatchMessage, then
perform a match.
                if(msg is StartMatchMessage)
                {
                    var info = msg as StartMatchMessage;

                    state = GameState.DoingMatch;
                    board.startMatch(info.xCon, info.oCon);
                    state = GameState.Waiting;
                }
                else if(msg is ThrowExceptionMessage) // Used
for testing the try-catch statement guarding this function.
                    throw new Exception();
                else // Otherwise, requeue it
                    this.gameQueue.Enqueue(msg);
            }
            else if(state == GameState.Crashed) // If an
exception is ever thrown, then some things have to be reset.
            {
                state = GameState.Waiting;
                board = new Board(); // Making sure the board
isn't in an invalid state.

                this.Dispatcher.Invoke(() =>
                {
                    this.updateText("[An error has occured]",
"[Please start a new match]");

                    // Stop the match if one is running.
                    if((this._flags & Flags.GameRunning) > 0)
                        this.onEndMatch();
                });
            }
        }
        catch (Exception ex) // Catch any exceptions, and let the
UI thread inform the user.
        {

```

```

        // If it's a thread exception, don't bother
reporting it.
        // This is because it's most likely an exception
telling the thread to close itself.
        if(ex is ThreadAbortException)
            return;

        this.Dispatcher.Invoke(() => this.reportException(ex));
        state = GameState.Crashed;
    }
}
}
```

Unittests/GameHash.cs

```

using Microsoft.VisualStudio.TestTools.UnitTesting;
using CS_Project.Game;
using System;
using System.Linq;
using System.IO;

using Hash = CS_Project.Game.Board.Hash;

namespace Unittests
{
    [TestClass]
    public class Game_Hash
    {
        [TestMethod]
        public void testHash()
        {
            var hash = new Hash(Board.Piece.X);

            // First, check that all 9 indicies can be used (and also test
isEmpty)
            foreach(int i in Enumerable.Range(0, (int)Board.pieceCount))
                Assert.IsTrue(hash.isEmpty(i));

            // Test setPiece and getPiece
            hash.setPiece(Board.Piece.Empty, 0);
            hash.setPiece(Board.Piece.O, 1);
            hash.setPiece(Board.Piece.X, 2);
            Assert.AreEqual(hash.getPiece(0), Board.Piece.Empty);
            Assert.AreEqual(hash.getPiece(1), Board.Piece.O);
            Assert.AreEqual(hash.getPiece(2), Board.Piece.X);

            // Test setPiece exception
            try
            {
                hash.setPiece(Board.Piece.X, 0, true); // Set the 0th
piece to x
                hash.setPiece(Board.Piece.O, 0, false); // Then try to
overwrite it, with the overwrite flag to set to false. Triggering the
exception.
                Assert.Fail("The exception for setPiece wasn't thrown.");
            }
            catch(HashException) { }

            // Test the enforceIndex exception
            try
            {
                hash.setPiece(Board.Piece.Empty, (int)Board.pieceCount);
                Assert.Fail("The exception for enforceIndex wasn't
thrown.");
            }
            catch(ArgumentOutOfRangeException) { }

            // Test isMyPiece
            hash.setPiece(hash.myPiece, 0, true);
            Assert.IsTrue(hash.isMyPiece(0));

```



```
// Test ToString
hash.setPiece(hash.myPiece, 0, true);
hash.setPiece(Board.Piece.Empty, 1, true);
hash.setPiece(hash.otherPiece, 2, true);
Assert.AreEqual(hash.ToString(),
$"{Hash.myChar}.{Hash.otherChar}.....");

// Test second constructor of Hash
hash = new Hash(Board.Piece.X, "MO.OM.MOM");

var mineIndicies = new int[]{0, 4, 6, 8};
var otherIndicies = new int[]{1, 3, 7};
var emptyIndicies = new int[]{2, 5};

Array.ForEach(mineIndicies, i => { Assert.IsTrue(hash.isMyPiece(i)); });
Array.ForEach(otherIndicies, i => { Assert.IsTrue(!hash.isMyPiece(i)); });
Array.ForEach(emptyIndicies, i => { Assert.IsTrue(hash.isEmpty(i)); });

// Test Clone and Equals
Assert.IsTrue(hash.Clone().Equals(hash));
}
```

```

[TestMethod()]
public void hashSerialiseTest()
{
    var dir = "Temp";
    var file = "Serialised_Hash.bin";
    var path = $"{dir}/{file}";

    if (!Directory.Exists(dir))
        Directory.CreateDirectory(dir);

    using (var stream = File.Create(path))
    {
        using (var writer = new BinaryWriter(stream))
        {
            var hash = new Hash(Board.Piece.X, "MO.OM.MOM");
            var hash2 = new Hash(Board.Piece.O, "OM.MO.OMO");

            hash.serialise(writer);
            hash2.serialise(writer);
        }
    }

    using (var stream = File.OpenRead(path))
    {
        using (var reader = new BinaryReader(stream))
        {
            var hash = new Hash();

            hash.deserialise(reader,
GameFiles.treeFileVersion);
            Assert.AreEqual("MO.OM.MOM", hash.ToString());
            Assert.AreEqual(Board.Piece.X, hash.myPiece);
            Assert.AreEqual(Board.Piece.O, hash.otherPiece);

            hash.deserialise(reader,
GameFiles.treeFileVersion);
            Assert.AreEqual("OM.MO.OMO", hash.ToString());
            Assert.AreEqual(Board.Piece.O, hash.myPiece);
            Assert.AreEqual(Board.Piece.X, hash.otherPiece);
        }
    }
}
}

```

UnitTests/AverageTests.cs

```

using Microsoft.VisualStudio.TestTools.UnitTesting;
using CS_Project.Game;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using Hash = CS_Project.Game.Board.Hash;

namespace CS_Project.Game.Tests
{
    [TestClass()]
    public class AverageTests
    {
        [TestMethod()]
        public void statisticallyBestTest()
        {
            var m    = Hash.myChar;
            var o    = Hash.otherChar;
            var p    = Board.Piece.X;
            var root = Node.root;
            root.children.AddRange(new Node[]
            {
                new Node(new Hash(p, $"{m}....."), 0, 6, 6), // 50% win
                new Node(new Hash(p, $"{m}....."), 1, 9, 3) // 75% win
            });

            root.children[0].children.AddRange(new Node[] // Adding to the 50% node
            {
                new Node(new Hash(p, $"{m}.{m}....."), 2, 6, 6), // 50% win
                (path average of 50%)
                new Node(new Hash(p, $"{m}..{m}....."), 3, 3, 9), // 25% win
                (path average of 62.5%)
            });

            root.children[1].children.AddRange(new Node[] // Adding to the 75% node
            {
                new Node(new Hash(p, $"{m}{m}....."), 0, 9, 3) // 75% win
                (path average of 75%)
            });

            var best = Average.statisticallyBest(root);
            Assert.IsTrue(best.averageWinPercent == 75.0f);
            Assert.IsTrue(best.path.Count == 2);
            Assert.IsTrue(best.path[0].hash.ToString() == $"{m}....."); // It
            should've chosen the 75% -> 75% path.
            Assert.IsTrue(best.path[1].hash.ToString() == $"{m}{m}.....");
        }
    }
}

```

UnitTests/BoardTests.cs

```

using CS_Project.Game;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using CS_Project.Game.Controllers;

namespace CS_Project.Game.Tests
{
    [TestClass]
    public class BoardTests
    {
        class NullController : Controller
        {
            int last = 0;

            public override void onAfterTurn(Board.Hash boardState, int index)
            {
                Assert.IsTrue(boardState.isMyPiece(this.last));
            }

            public override void onDoTurn(Board.Hash boardState, int index)
            {
                // Put a piece in any empty slot, that isn't on the first
row.
                for (var i = 3; i < Board.pieceCount; i++)
                {
                    if (boardState.isEmpty(i))
                    {
                        base.board.set(i, this);
                        this.last = i;
                        break;
                    }
                }
            }
        }
    }
}

```

```

class StupidController : Controller
{
    public override void onAfterTurn(Board.Hash boardState, int index)
    {
        var str =
boardState.ToString().Replace(Board.Hash.otherChar, Board.Hash.emptyChar);

        Assert.IsTrue(str == "M....." || str == "MM....." ||
str == "MMM.....");
    }

    public override void onDoTurn(Board.Hash boardState, int index)
    {
        for (var i = 0; i < 3; i++)
        {
            if (boardState.isEmpty(i))
            {
                base.board.set(i, this);
                break;
            }
        }
    }
}

[TestMethod]
public void basicMatchTest()
{
    // This should be improved at some point.
    var board = new Board();
    board.startMatch(new NullController(), new StupidController());
}
}

```

UnitTests/GameFilesTests.cs

```

using Microsoft.VisualStudio.TestTools.UnitTesting;
using CS_Project.Game;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CS_Project.Game.Tests
{
    [TestClass()]
    public class GameFilesTests
    {
        [TestMethod()]
        public void treeExistsTest ()
        {
            const string name = "Test_dummy";

            Assert.IsFalse(GameFiles.treeExists(name));
            GameFiles.saveTree(name, Node.root);
            Assert.IsTrue(GameFiles.treeExists(name));

            GameFiles.removeTree(name);
        }

        [TestMethod()]
        public void removeTreeTest ()
        {
            const string name = "Test_remove";

            Assert.IsFalse(GameFiles.treeExists(name));
            GameFiles.saveTree(name, Node.root);

            Assert.IsTrue(GameFiles.treeExists(name));
            GameFiles.removeTree(name);

            Assert.IsFalse(GameFiles.treeExists(name));

            // Check for exception
            try
            {
                GameFiles.removeTree(name, true);
                Assert.Fail("No exception was thrown.");
            }
            catch(Exception ex) { }
        }
    }
}

```

```

[TestMethod()]
public void saveTreeLoadTreeTest()
{
    const string name = "Test_saveAndLoad";
    const uint wins = 20;
    const uint losses = 30;

    // Create a simple tree
    var root = Node.root;
    var piece = Board.Piece.X;
    root.children.Add(new Node(new Board.Hash(piece, "M....."),
0, wins, losses));
    root.children.Add(new Node(new Board.Hash(piece, "O....."),
0, losses, wins));
    root.children[0].children.Add(new Node(new Board.Hash(piece,
"MO....."), 1, wins, losses));

    // Save it
    GameFiles.saveTree(name, root);

    // Then load it back in, and see if the nodes are still correct.
    root = GameFiles.loadTree(name);

    var node = root.children[0];
    Assert.IsTrue(node.hash.ToString() == "M.....");
    Assert.IsTrue(node.index == 0);
    Assert.IsTrue(node.won == wins);
    Assert.IsTrue(node.lost == losses);

    node = root.children[1];
    Assert.IsTrue(node.hash.ToString() == "O.....");
    Assert.IsTrue(node.index == 0);
    Assert.IsTrue(node.won == losses);
    Assert.IsTrue(node.lost == wins);

    node = root.children[0].children[0];
    Assert.IsTrue(node.hash.ToString() == "MO.....");
    Assert.IsTrue(node.index == 1);
    Assert.IsTrue(node.won == wins);
    Assert.IsTrue(node.lost == losses);

    // And now to quickly check that some exceptions get thrown
    try
    {
        GameFiles.saveTree(name, root, false); // Cannot
overwrite existing tree.
        Assert.Fail("No exception was thrown.");
    }
    catch (Exception) { }

    try
    {
        GameFiles.saveTree("s", null); // Cannot pass a null root
node.
        Assert.Fail("No exception was thrown.");
    }
    catch (Exception) { }

    Assert.IsNull(GameFiles.loadTree("", false));
}

```

```
        try
        {
            GameFiles.loadTree(""); // No tree named ""
            Assert.Fail("No exception was thrown.");
        }
        catch(Exception) { }
    }
}
```


UnitTests/NodeTests.cs

```

using CS_Project.Game;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System.IO;

using Hash = CS_Project.Game.Board.Hash;
using System;

namespace CS_Project.Game.Tests
{
    [TestClass()]
    public class NodeTests
    {
        [TestMethod()]
        public void walkTest()
        {
            var m = Hash.myChar;
            var o = Hash.otherChar;
            var p = Board.Piece.X;

            var root = Node.root;
            root.children.AddRange(new Node[]
            {
                new Node(new Hash(p, $"{m}....."), 0),
                new Node(new Hash(p, $"{o}....."), 1)
            });
            root.children[0].children.Add(new Node(new Hash(p, $"{m}.{o}....."),
2));

            // Visualisation of what 'tree' looks like
            /*
               /-----[0]-----/ "M.O....."
            *   /-----[0]-----/ "M....."
            * root
            *   /-----[1]-----/ ".O....."
            * */

            // This action is used with Node.walk, it will keep track of the last
            node it visited.
            Node last = null;
            Action<Node> getLast = (node => last = node);

            // First, seeing if it returns false on an invalid path.
            // "M....." -> "MM....."
            var path = new Hash[] { new Hash(p, $"{m}....."),
                                   new Hash(p, $"{m}{m}.....") };
            Assert.IsFalse(root.walk(path, getLast)); // "walk" returns false if
            the entire path couldn't be followed.
            Assert.IsTrue(last == root.children[0]); // Confirm that the only
            node we walked to was "M....."

            // Then see if depth works
            Assert.IsTrue(root.walk(path, getLast, 1)); // It will return true
            now, since we walked 'depth' amount of nodes sucessfully
            Assert.IsTrue(last == root.children[0]);

            // Then finally see if it walks through an entire path properly
            // We have to change the last hash in path first though
            path[1] = new Hash(p, $"{m}.{o}....."); // "M....." ->
            "M.O....."

```

```

        Assert.IsTrue(root.walk(path, getLast)); // If this
returns true, then the entire path was walked
        Assert.IsTrue(last == root.children[0].children[0]); // Then we make
sure the last node walked to is correct.
    }

[TestMethod()]
public void nodeSerialiseTest()
{
    var m = Hash.myChar;
    var o = Hash.otherChar;
    var dir = "Temp";
    var file = "Serialised_Node.bin";
    var path = $"{dir}/{file}";

    // Make sure the temp directory exists.
    if (!Directory.Exists(dir))
        Directory.CreateDirectory(dir);

    // Then, write out a simple tree to a file.
    using (var stream = File.Create(path))
    {
        using (var writer = new BinaryWriter(stream))
        {
            var root = Node.root;
            root.children.AddRange(new Node[]
            {
                new Node(new Hash(Board.Piece.X, $"{m}....."),
0, 3, 5),
                new Node(new Hash(Board.Piece.X, $"{m}....."),
1, 1, 4)
            });
            root.children[0].children.Add(new Node(new
Hash(Board.Piece.O, $"{m}{o}....."), 1, 2, 3));

            /* Tree:
            *
            *                                     /---[0]---/ "MO....."
            *                               /---[0]---/ "M....."
            * "....."(root)
            *                               /---[1]---/ ".M....."
            * */

            root.serialise(writer);
        }
    }

    // Then, read the tree back in from the file, and confirm *every*
node is correct.
    using (var stream = File.OpenRead(path))
    {
        using (var reader = new BinaryReader(stream))
        {
            var root = Node.root;
            root.deserialise(reader, GameFiles.treeFileVersion);

            var n = root.children[0]; // Current node we're asserting
Assert.AreEqual(new Hash(Board.Piece.X, $"{m}....."),
n.hash);

            Assert.AreEqual(0u, n.index);
            Assert.AreEqual(3u, n.won);
            Assert.AreEqual(5u, n.lost);

```

```

        n = root.children[1];
        Assert.AreEqual(new Hash(Board.Piece.X, $"{m}....."),
n.hash);

        Assert.AreEqual(1u, n.index);
        Assert.AreEqual(1u, n.won);
        Assert.AreEqual(4u, n.lost);

        n = root.children[0].children[0];
        Assert.AreEqual(new Hash(Board.Piece.O, $"{m}{o}....."),
n.hash);

        Assert.AreEqual(1u, n.index);
        Assert.AreEqual(2u, n.won);
        Assert.AreEqual(3u, n.lost);
    }
}

[TestMethod()]
public void mergeTest()
{
    var m = Hash.myChar;
    var o = Hash.otherChar;

    // This is the tree that's going to be merged into.
    var destination = Node.root;

    // "M....." (W:2, L:1) -> [0] "MO....." (W:1 L:0)
    //                               -> [1] "M.O....." (W:1 L:1)
    // ".M....." (W:0 L:0)
    var source = Node.root;
    source.children.Add(new Node(new Hash(Board.Piece.X,
    $"{m}....."), 0, 2, 1));
    source.children.Add(new Node(new Hash(Board.Piece.X,
    $"{m}....."), 1, 0, 0));
    source.children[0].children.Add(new Node(new Hash(Board.Piece.X,
    $"{m}{o}....."), 1, 1, 0));
    source.children[0].children.Add(new Node(new Hash(Board.Piece.X,
    $"{m}.{o}....."), 2, 1, 1));

    // Doing it twice to make sure it works for nodes that exist, and
    ones that don't exist.
    Node.merge(destination, source);
    Node.merge(destination, source);

    var node = destination.children[0];
    Assert.IsTrue(node.hash.ToString() == $"{m}.....");
    Assert.IsTrue(node.index == 0);
    Assert.IsTrue(node.won == 4);
    Assert.IsTrue(node.lost == 2);

    node = destination.children[1];
    Assert.IsTrue(node.hash.ToString() == $"{m}.....");
    Assert.IsTrue(node.index == 1);
    Assert.IsTrue(node.won == 0);
    Assert.IsTrue(node.lost == 0);

    node = destination.children[0].children[0];
    Assert.IsTrue(node.hash.ToString() == $"{m}{o}.....");
    Assert.IsTrue(node.index == 1);
    Assert.IsTrue(node.won == 2);

```

```
Assert.IsTrue(node.lost == 0);

node = destination.children[0].children[1];
Assert.IsTrue(node.hash.ToString() == $"{m}.{o}.....");
Assert.IsTrue(node.index == 2);
Assert.IsTrue(node.won == 2);
Assert.IsTrue(node.lost == 2);
}
}
}
```

Pseudocode/CalculateAverages.txt

[This is the original pseudocode for what eventually was modified to become the 'WalkPaths' and 'StatisticallyBest' algorithm]

```

refType Node
    var Wins      as UnsignedInteger
    var Losses    as UnsignedInteger
    var Children  as DynamicArray:Node

    function AverageWinPercentage() returns FloatingPoint
        return (self.Wins / (self.Wins + self.Losses)) * 100
    end
end

valueType Average
    var Path      as DynamicArray:Node
    var Percentage as FloatingPoint

    // Calculates the average win percentage of the path.
    function CalculateFinalPercentage() returns Void
        self.Percentage = (self.Percentage / (self.Path.Length *
100.0)) * 100
    end
end

function CalculateAverages(Root as Node) returns DynamicArray:Average
    var Averages as DynamicArray:Average

    closure Walk(CurrentNode as Node, CurrentAverage as Average)
        CurrentAverage.Percentage += CurrentNode.AverageWinPercentage()
        CurrentAverage.Path       = CurrentAverage.Path.Copy()
        CurrentAverage.Path.Add(CurrentNode)

    // Slight note: Because "Average" is a valueType, every copy of it will
    // still be using
    // the same DynamicArray, which will cause issues. Which is why it has to
    // be copied before we add to it.
    // This is of course, inefficient in terms of memory usage. But my entire
    // algorithm is probably inefficient.

        if CurrentAverage.Children.Length == 0 then
            CurrentAverage.CalculateFinalPercentage()
            Averages.Add(CurrentAverage)
        else
            foreach Child as Node, CurrentNode.Children do
                Walk(Child, CurrentAverage)
            end
        end
    end

    var StartAverage as Average
    Walk(Root, StartAverage)

    return Averages
end

```

[Before I even touched any code for the project (and back when I was going to use Python) this is the document I was working on to explain how the hashing and AI worked. There is no particular reason I have included this other than to offer a view of how I used to think the AI would work, compared to how it works now.]

Overview

The basic idea for this project is an AI that can play tic-tac-toe, but it will save certain information about each match to best predict how it could win.

[Details] How will the AI work?

How the game board is hashed

Before we can talk about the details of how the AI stores its data, and how it uses it during a match, we need to go over how to represent the game board in a way we can use it to store data.

The game board will be represented by a 9-character string where each character represents a slot on the board, with the first 3 characters representing the top row (first 3 slots), the next 3 characters representing the middle row (next 3 slots), and the last 3 characters representing the bottom row (last 3 slots).

If the character is ".", then neither the AI or the other player have placed one of their pieces there.

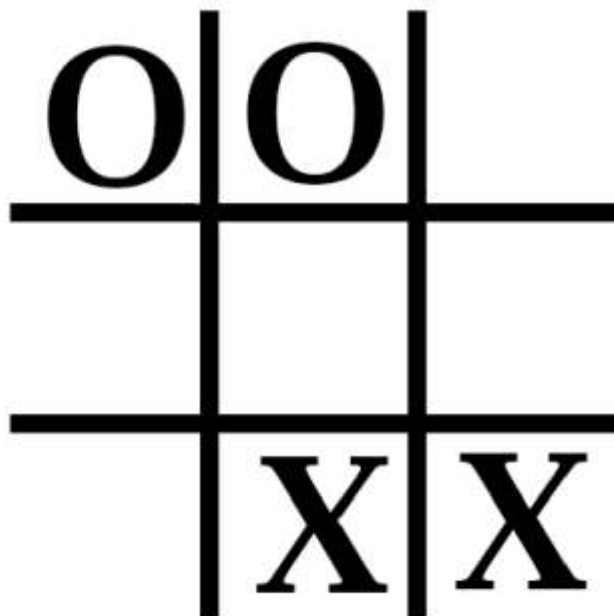
If the character is "M", then the AI has placed a piece there. The "M" stands for "Me".

If the character is "O", then the other player has placed a piece there. The "O" stands for "Other".

The benefit of storing which player has a piece there, rather than just saying "This slot had an X, and the other slot had an O" is that it makes the data independent of what piece the AI is. If the board wasn't stored like this, then the data between whether the AI was playing as X or if it was playing as O would be different and couldn't be shared between them (although, this may actually be desirable if either X or O has some kind of advantage the AI could take note of, so this detail may change). A side benefit of this is that the AI can be put up against itself while sharing the same set of data.

As an example, examine the game board below, and see how the hash of it looks.

The AI is X, and the other player is O.



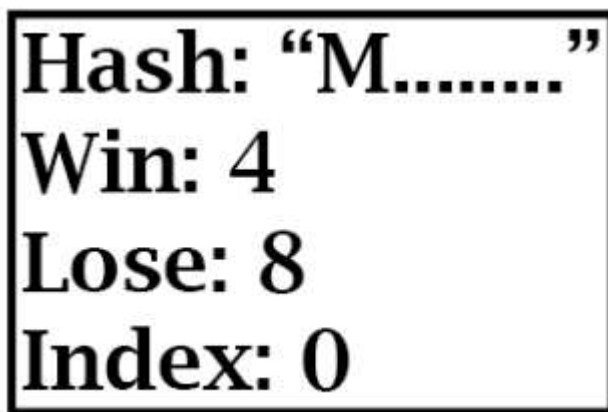
This game board will be hashed as
"OO.....MM".

How is the data from each match stored?

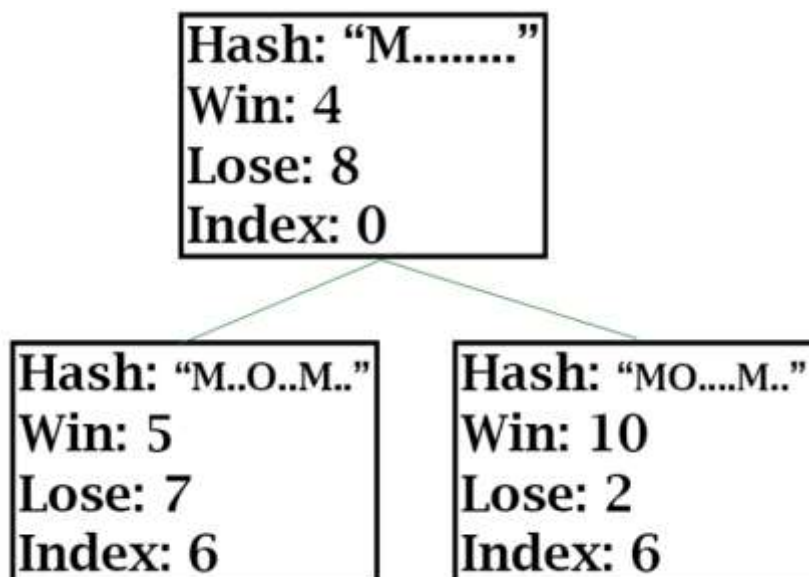
The data will be stored as a tree. Where each node represents the state of a board after the AI has made a move. The node will contain:

- The hash of the board after the AI made its move.
- A “Win” counter, which describes how many times the AI has won when doing the move it did.
- A “Lose” counter to compliment the “Win” counter.
- An “Index” of the slot the AI placed its piece in. While this *can* be calculated easily by comparing the previous board hash of the current node to its parent’s state, it’s needlessly complex. This is 0 based, so for example, if the AI placed a piece in the top-left corner, the index would be 0, and the bottom-right corner would be 8.
- A list containing all of the node’s “children”. This is all of the moves the AI has attempted to make after making the move that the current node represents.

Here is a visualisation of a single node, where the AI has placed its piece at the top-left corner.



And this is an example of a slightly more fleshed out tree of nodes.



As can be seen in the children nodes, even though the AI made the exact same move (placing at index 6), the other player made different moves, which allows different “paths” the AI could look through for the optimal move.

How will the AI behave during a match?

Each section will first contain a more higher-level explanation, and then one that goes into details on how I plan to implement things.

During a match [High-level]

The AI will be configurable, in the sense that it will have different “modes” which will tell it how it should try and figure out a move. The simple overview for each mode I plan for is:

- The “StatisticallyBest” mode – Where each move, the AI will look in the Global move tree to see what all the paths it’s learned about are, and choose the one that, statistically, is most likely to win.
- The “StatisticallyLikely” mode – Which is similar to the previous mode, except it will look at the top 3 paths that are most likely to win, using the same method as StatisticallyBest, and then randomly choosing one of the paths.
- The “RandomExisting” mode – Where the AI will look at all existing paths it can currently take, and just pick one randomly. This will mostly be used as a fallback for other algorithms.
- The “RandomNotExisting” mode – Which is similar to RandomExisting except it will pick a random path it *hasn’t yet taken*. Again, used mostly as a fallback.
- The “RandomAny” mode – Which will just pick any random path it can take regardless if it’s taken it before or not.

Now, during the AI’s turn it will execute the logic of its selection mode, while adding the move it made to its Local move tree. This will then be repeated until the game is over.

At the end of a match [High-level]

During a match, after every move the AI takes, it’ll build its own tree of moves (which looks like the tree from before, but it’s just the single path the AI takes) which consists of the moves the AI decided to take.

After a match, the AI will then update the “Global” move tree, which contains the data of every match the AI has played. The AI will update the Global move tree using the “Local” move tree it created during the match, and to do this, it’ll basically “walk” it’s Local tree and mirror the results into the Global move tree. So, if the AI lost for example, it’ll walk the path it took in the Local tree and update the node in the Global tree by bumping the “Lost” counter up by one.

During a match [Technical]

These are the main classes that are going to be driving the game:

- The “Controller” class, which is an abstract class that is used for code to define a way for something to interact with a game board. The reason an abstract approach was chosen, is so it’s very easy to go between “Player vs AI”, “AI vs AI”, “Player vs Player”, etc.
- The “Player” class, which inherits from Controller and is responsible for letting the player interact with the board.
- The “AI” class, which inherits from Controller and is responsible for performing the AI’s logic.
- The “SelectMode” class, which is an abstract class that is attached to an “AI” class to control how it selects its next move during a match. These modes are explained in detail later on.
- The “MoveTree” class, which is used to manipulate a move tree, including functions to serialise/unserialise the data to/from a file, and additional helper functions to easily manipulate the MoveTree with another MoveTree (For example, the AI could possibly use the code ``globalMoveTree.updateData(tree=localTree, didAIWin=False)`` to tell the Global move tree to use “localTree” to bump the “Lose” counter of every node in the localTree’s path).
- The “Board” class, which will contain the game board state. It will have two Controllers, one for X and one for O. It is responsible for carrying out turns, and providing an interface for the controllers to manipulate and receive the hash of the board.
- The “GUI” class, which is used to display the state of the game board to the user. Some classes, such as the Player class, may make use of the GUI class to allow the user to interact with the GUI to manipulate the game board.

At the start of the match, the AI will create its Local MoveTree class.

During the match, and when it’s the AI’s turn, it will take the following steps:

1. Allow the AI’s Controller to perform its logic, and to perform a move.
2. Using the Local move tree, the AI will create a node based on the move the controller made.

The algorithms of each SelectMode are as followed(It should be noted none of the algorithms have been tested yet):

RandomAny

1. Get the hash of the current state of the board
2. Generate a random number between 0 and 8 (inclusive)
3. Using the number as an index for the board's hash, check if the index points to '.'
 - a. If the index points to a '.', then place the AI's piece at that position.
 - b. If the index doesn't point to a '.', then go back to 3.

RandomExisting

1. Follow the Local move tree and the Global move tree at the same time.
 - a. If at some point, the Global tree cannot be followed exactly like the Local move tree, resort to using RandomAny.
2. After walking through the trees, select the children from the node in the Global move tree.
3. If the node has no children, resort to RandomNotExisting for the match.
4. Generate a random number between 0 and how many children there are (inclusive).
5. Using the number as an index for the list of children, get the child at the index.
6. Place the AI's piece at the same slot as the child's "index" member.

RandomNotExisting

1. Follow the steps for RandomExisting, until step 3 is reached (don't perform step 3)
2. Perform the entire algorithm for RandomAny, but instead of placing the AI's piece, replace '.' with 'M'.
3. Check if the newly modified hash matches the hash of any of the children nodes.
 - a. If there is a match, go back to 2.
 - b. If there is no match, use the number generated during RandomAny's algorithm, and use this number as the slot to place the AI's piece.

StatisticallyBest

The algorithm to calculate the average of a path is stored in “FindAverage_PseudoCode.txt” alongside this file. The “Average” type is also described there.

1. Before “CalculateAverages” is called, a “Root” must be found.
 - a. Follow the Local move tree and the Global move tree at the same time.
 - b. If the Global move tree cannot be followed exactly like the Local move tree, resort to using RandomAny for the remainder of the match.
 - c. Otherwise, once the Local move tree has been followed completely (In both the Local and Global move trees) select the node in the Global move tree as the Root.
 - d. If the Local move tree contains no nodes, perform the RandomExisting algorithm for one turn.
2. Using the new Root node, call CalculateAverages on it. The returned array of averages will be known as “Averages”.
3. If the Averages array has a length of 0, then resort to RandomAny for the remainder of the match.
4. Sort the Averages array, where the 0th Average in the array has the largest Percentage, and the last Average has the lowest Percentage.
5. Select the 0th Average, and select it’s 0th Path node, then place the AI’s piece at the same slot as the Path node’s ‘index’ member.

StatisticallyLikely

1. Follow the steps for StatisticallyBest up to and including step 4.
2. Generate a random number between 0 and n , where n is the smaller number between 3 and the Averages array’s length.
3. Use this number as an index to select one of the ‘Average’s in the Averages array.
4. Then, select the 0th node in the Average’s “Path”, and place the AI’s piece at the same slot as the node’s ‘index’ member.

At the end of a match [Technical]

Once the match is over, the AI will apply the following algorithm to update the Global move tree:

1. Follow the Local move tree and the Global move tree at the same time.
 - a. If the Global move tree cannot be followed exactly like the Local move tree, then add the missing nodes into the Global move tree, making sure to bump their “Win” and “Loss” counter as appropriate.
 - b. Otherwise, for every node that is walked to in the Global move tree, bump its “Win” and “Loss” counter as appropriate.