Table of contents

[Table of contents 1](#_Toc474039093)

[Analysis 2](#_Toc474039094)

[Description 2](#_Toc474039095)

[Target Audience/Stakeholders 2](#_Toc474039096)

[Why a computer is suitable for the task 2](#_Toc474039097)

[Research 3](#_Toc474039098)

[Features and limitations 4](#_Toc474039099)

[Requirements 4](#_Toc474039100)

[Success Criteria 5](#_Toc474039101)

[Design 6](#_Toc474039102)

[Decomposition of the problem 6](#_Toc474039103)

[Proposed structure of the program 6](#_Toc474039104)

[Algorithms 6](#_Toc474039105)

[Usability 6](#_Toc474039106)

[Key variables and data structures 6](#_Toc474039107)

[Test Data for development 6](#_Toc474039108)

[Test Data for beta testing 6](#_Toc474039109)

[Development 6](#_Toc474039110)

[Iterations of development 6](#_Toc474039111)

[Prototypes 6](#_Toc474039112)

[Evidence of modular code 6](#_Toc474039113)

[Evidence of validation 6](#_Toc474039114)

[Review 6](#_Toc474039115)

[Evaluation 7](#_Toc474039116)

[Testing 7](#_Toc474039117)

[Testing of usability features 7](#_Toc474039118)

[Overall evaluation 7](#_Toc474039119)

[Future Maintenance 7](#_Toc474039120)

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 actual 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 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.

[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 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

[TEMPORARY: Grid template]

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

# 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:

|  |  |  |
| --- | --- | --- |
| **X** |  |  |
|  |  |  |
|  |  |  |

|  |  |  |
| --- | --- | --- |
| X |  |  |
|  | **O** |  |
|  |  |  |

|  |  |  |
| --- | --- | --- |
| X | **O** |  |
|  |  |  |
|  |  |  |

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.

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:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | |  |  |  | | --- | --- | --- | |  |  |  | |  |  |  | |  |  |  | |  |
|  | |  |  |  | | --- | --- | --- | | **X** |  |  | |  |  |  | |  |  |  | |  | |  |  |  | | --- | --- | --- | |  |  |  | |  |  |  | |  | **X** |  | |
| |  |  |  | | --- | --- | --- | | X |  |  | |  |  | **O** | |  |  |  | |  | |  |  |  | | --- | --- | --- | | X |  |  | |  |  |  | |  | **O** |  | | |  |  |  | | --- | --- | --- | |  |  |  | |  | **O** |  | |  | X |  | |
|  |  | |  |  |  | | --- | --- | --- | | X |  |  | |  |  | **X** | |  | O |  | | |  |  |  | | --- | --- | --- | |  |  |  | |  | O |  | | **X** | X |  | |

## 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 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’s 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, there is no way to determine which piece the AI is using, and which piece the player is using. 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..OM”.

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

It’s worth noting that, even if it’s very 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 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 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.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | [Root]  W:0 L:0 I:0   |  |  |  | | --- | --- | --- | |  |  |  | |  |  |  | |  |  |  | |  |
|  | W:8 L:4 I:1   |  |  |  | | --- | --- | --- | | **X** |  |  | |  |  |  | |  |  |  | |  | W:3 L:7 I:7   |  |  |  | | --- | --- | --- | |  |  |  | |  |  |  | |  | **X** |  | |
| W:1 L:1 I:5   |  |  |  | | --- | --- | --- | | X |  |  | |  |  | **O** | |  |  |  | |  | W:7 L:3 I:7   |  |  |  | | --- | --- | --- | | X |  |  | |  |  |  | |  | **O** |  | | W:3 L:7 I:4   |  |  |  | | --- | --- | --- | |  |  |  | |  | **O** |  | |  | X |  | |
|  |  | W:7 L:3 I:5   |  |  |  | | --- | --- | --- | | X |  |  | |  |  | **X** | |  | O |  | | W:3 L:7 I:6   |  |  |  | | --- | --- | --- | |  |  |  | |  | O |  | | **X** | X |  | |

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

A path is 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 show 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 x 100 = 25%).

Path #1

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | [Root]  W:0 L:0 I:0 W%:0   |  |  |  | | --- | --- | --- | |  |  |  | |  |  |  | |  |  |  | |  |
|  | W:8 L:4 I:1   |  |  |  | | --- | --- | --- | | **X** |  |  | |  |  |  | |  |  |  | |  | W:3 L:7 I:7  W%: 30% (0.3)   |  |  |  | | --- | --- | --- | |  |  |  | |  |  |  | |  | **X** |  | |
| W:1 L:1 I:5   |  |  |  | | --- | --- | --- | | X |  |  | |  |  | **O** | |  |  |  | |  | W:7 L:3 I:7   |  |  |  | | --- | --- | --- | | X |  |  | |  |  |  | |  | **O** |  | | W:3 L:7 I:4  W%: 30% (0.3)   |  |  |  | | --- | --- | --- | |  |  |  | |  | **O** |  | |  | X |  | |
|  |  | W:7 L:3 I:5   |  |  |  | | --- | --- | --- | | X |  |  | |  |  | **X** | |  | O |  | | W:3 L:7 I:6  W%: 30% (0.3)   |  |  |  | | --- | --- | --- | |  |  |  | |  | O |  | | **X** | X |  |   Overall W%:  0.3 x 100 = 30% |

Path #2

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | [Root]  W:0 L:0 I:0   |  |  |  | | --- | --- | --- | |  |  |  | |  |  |  | |  |  |  | |  |
|  | W:8 L:4 I:0  W%: 67% (0.67)   |  |  |  | | --- | --- | --- | | **X** |  |  | |  |  |  | |  |  |  | |  | W:3 L:7 I:7   |  |  |  | | --- | --- | --- | |  |  |  | |  |  |  | |  | **X** |  | |
| W:1 L:1 I:5   |  |  |  | | --- | --- | --- | | X |  |  | |  |  | **O** | |  |  |  | |  | W:7 L:3 I:7  W%: 70% (0.7)   |  |  |  | | --- | --- | --- | | X |  |  | |  |  |  | |  | **O** |  | | W:3 L:7 I:4   |  |  |  | | --- | --- | --- | |  |  |  | |  | **O** |  | |  | X |  | |
|  |  | W:7 L:3 I:5  W%: 70% (0.7)   |  |  |  | | --- | --- | --- | | X |  |  | |  |  | **X** | |  | O |  |   = 69% | W:3 L:7 I:6   |  |  |  | | --- | --- | --- | |  |  |  | |  | O |  | | **X** | X |  | |

Path #3

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | [Root]  W:0 L:0 I:0   |  |  |  | | --- | --- | --- | |  |  |  | |  |  |  | |  |  |  | |  |
|  | W:8 L:4 I:0  W%: 67% (0.67)   |  |  |  | | --- | --- | --- | | **X** |  |  | |  |  |  | |  |  |  | |  | W:3 L:7 I:7   |  |  |  | | --- | --- | --- | |  |  |  | |  |  |  | |  | **X** |  | |
| W:1 L:1 I:5  W%: 50% (0.5)   |  |  |  | | --- | --- | --- | | X |  |  | |  |  | **O** | |  |  |  |   = 58.5% |  | W:7 L:3 I:7   |  |  |  | | --- | --- | --- | | X |  |  | |  |  |  | |  | **O** |  | | W:3 L:7 I:4   |  |  |  | | --- | --- | --- | |  |  |  | |  | **O** |  | |  | X |  | |
|  |  | W:7 L:3 I:5   |  |  |  | | --- | --- | --- | | X |  |  | |  |  | **X** | |  | O |  | | W:3 L:7 I:6   |  |  |  | | --- | --- | --- | |  |  |  | |  | O |  | | **X** | X |  | |

As promised earlier, I will provide the algorithm 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.

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,
   1. Perform *Action* on *Path*.
   2. Go to Step 4.
3. Otherwise, get the *Node*’s next child.
   1. 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).
   2. Remove the last node in the *Path* (this is the child node we just got, and would’ve been added by step 1).
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 may specify an *Action* because of the reusability of this algorithm.

## 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.

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.*
   1. Set *BestPercentage* to *Percentage*.
   2. 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, 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).

## 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. Now, if the WalkPaths algorithm is used, where its *Action* is the StatisticallyBest algorithm, then the AI would choose to go down Path #2 (the diagram shown earlier).

So, the AI has chosen Path #2, so 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 ‘Match following path’ scenario.

If the player places their piece in slot 5 as the second move however (take a look at Path #3) then the match has gone off track from the path the AI selected, but the Global tree has a node for the move the player has chosen, 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.

# Proposed structure of the program

[TODO]

# Usability

[TODO]

# Key variables and data structures

[TODO]

# Test Data for development

[TODO]

# Test Data for beta testing

[TODO]

Development

# Iterations of development

[TODO] (Self note, remember to use Git to ‘go back in time’ if I need something like a screenshot from an earlier version)

# Prototypes

[TODO]

# Evidence of modular code

[TODO]

# Evidence of validation

[TODO]

# Review

[TODO]

Evaluation

# Testing

[TODO]

# Testing of usability features

[TODO]

# Overall evaluation

[TODO]

# Future Maintenance

[TODO]