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).

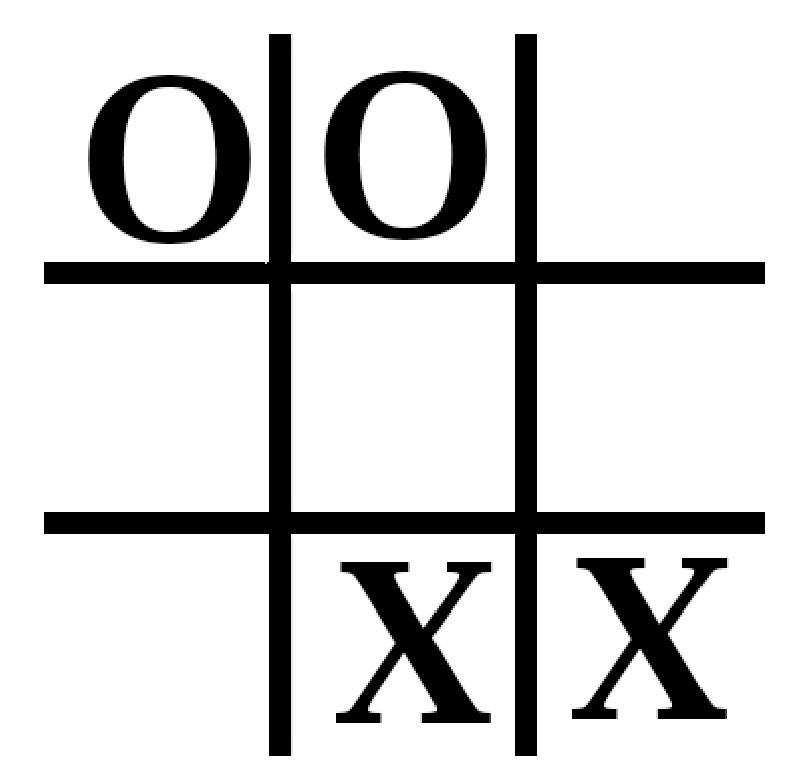
If the character is “.”, then neither the AI or the other player have placed one of their pieces there.

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.

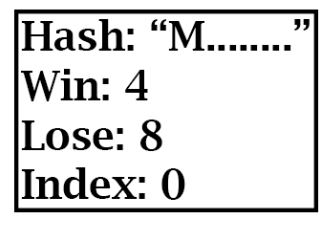
This game board will be hashed as “OO…..MM”.

# How is the data from each match stored?

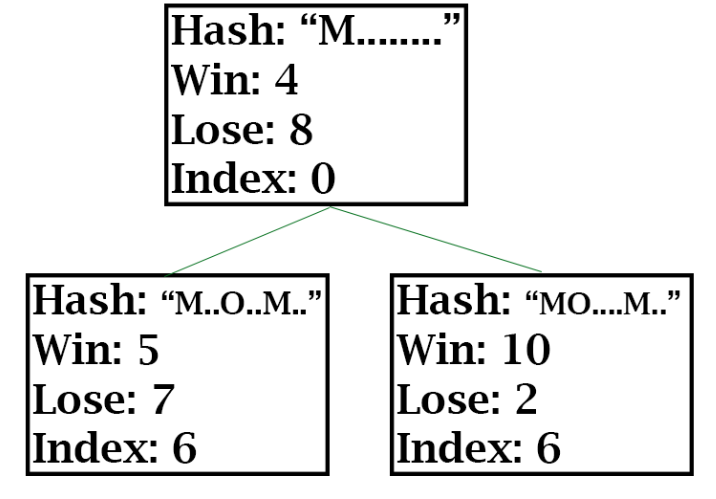
The data will be stored as a tree. Where each node represents the state of a board after the AI has made a move. The node will contain:

* 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 ‘.’
   1. If the index points to a ‘.’, then place the AI’s piece at that position.
   2. 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.
   1. 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.
   1. If there is a match, go back to 2.
   2. 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.
   1. Follow the Local move tree and the Global move tree at the same time.
   2. If the Global move tree cannot be followed exactly like the Local move tree, resort to using RandomAny for the remainder of the match.
   3. 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.
   4. 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.
   1. 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.
   2. Otherwise, for every node that is walked to in the Global move tree, bump its “Win” and “Loss” counter as appropriate.