Ultimate Tic Tac Toe AI

Contents

[Introduction: 4](#_Toc31328972)

[Background Research: 6](#_Toc31328973)

[Competing Products: 6](#_Toc31328974)

[Key Features: 7](#_Toc31328975)

[Rule Variation: 7](#_Toc31328976)

[AI: 8](#_Toc31328977)

[Issues and Complexity 8](#_Toc31328978)

[Monte Carlo Tree Search Algorithm: 8](#_Toc31328979)

[UCT Equation: 10](#_Toc31328980)

[Monte Carlo Tree Search- Advantages & Disadvantages: 11](#_Toc31328981)

[Data Flow Diagrams and Objectives: 12](#_Toc31328982)

[Main Game: 12](#_Toc31328983)

[1-Main Menu: 12](#_Toc31328984)

[2-Tutorial Page: 12](#_Toc31328985)

[3-Play Game Menu: 13](#_Toc31328986)

[4-Change AI Difficulty Menu: 13](#_Toc31328987)

[5-Initialise Game: 13](#_Toc31328988)

[6-AI: 13](#_Toc31328989)

[7-Human vs AI mode: 13](#_Toc31328990)

[8-Local Multiplayer Mode: 14](#_Toc31328991)

[9-Display Game: 14](#_Toc31328992)

[10-User Make Move/Input: 14](#_Toc31328993)

[AI Analyse and Make Move: 14](#_Toc31328994)

[11-Validate Movement: 15](#_Toc31328995)

[12-Check Win: 15](#_Toc31328996)

[13-End Screen: 15](#_Toc31328997)

[14-Save Game/Load Game: 15](#_Toc31328998)

[AI Data Flow Diagram and Objectives: 16](#_Toc31328999)

[15-Initialise Game Tree 16](#_Toc31329000)

[16-Select Node 16](#_Toc31329001)

[17-Expand 16](#_Toc31329002)

[18-Simulate 16](#_Toc31329003)

[19-Back-Propagate 16](#_Toc31329004)

[20-Make Move 17](#_Toc31329005)

[21-Discard Game Tree 17](#_Toc31329006)

[22-C Constant Selection 17](#_Toc31329007)

[Modelling 17](#_Toc31329008)

[Monte Carlo Algorithm Pseudocode Model 17](#_Toc31329009)

[Monte Carlo Algorithm Python Model 18](#_Toc31329010)

[Project Technology 23](#_Toc31329011)

[Game Rules: 23](#_Toc31329012)

[File Structure and Organisation 24](#_Toc31329013)

[Class Diagram 24](#_Toc31329014)

[Classes 24](#_Toc31329015)

[Node Class: 24](#_Toc31329016)

[MonteCarlo Class: 24](#_Toc31329017)

[Game Class: 25](#_Toc31329018)

[Menu Class 25](#_Toc31329019)

[Key Data Structures: 25](#_Toc31329020)

[Node.state 25](#_Toc31329021)

[Node.parent 25](#_Toc31329022)

[Node.children 25](#_Toc31329023)

[Node.value 25](#_Toc31329024)

[Node.prev\_move and Node.current\_player 26](#_Toc31329025)

[Node.possible\_moves 26](#_Toc31329026)

[Node.depth 26](#_Toc31329027)

[Node.UCT 26](#_Toc31329028)

[Game Tree: 27](#_Toc31329029)

[Game I/O 27](#_Toc31329030)

[Node Display 27](#_Toc31329031)

[Node.display\_node() 28](#_Toc31329032)

[Move Input 28](#_Toc31329033)

[Function: input\_convertor() 29](#_Toc31329034)

[Key Functions 30](#_Toc31329035)

[Node Class: 30](#_Toc31329036)

[Global Functions: 30](#_Toc31329037)

[Function: opposite() 31](#_Toc31329038)

[Function check\_win() 31](#_Toc31329039)

[Function: check\_move() 32](#_Toc31329040)

[Function: get\_valid\_moves 32](#_Toc31329041)

[Monte Carlo Class 33](#_Toc31329042)

[MonteCarlo.\_\_init\_\_(self) 33](#_Toc31329043)

[MonteCarlo.create\_child\_node 34](#_Toc31329044)

[MonteCarlo.get\_UCT() 34](#_Toc31329045)

[MonteCarlo.select() 35](#_Toc31329046)

[MonteCarlo.simulation() 35](#_Toc31329047)

[MonteCarlo.simulate() 36](#_Toc31329048)

[MonteCarlo.expand() 36](#_Toc31329049)

[MonteCarlo.back\_propogate() 36](#_Toc31329050)

[MonteCarlo.make\_move() 36](#_Toc31329051)

[MonteCarlo.MonteCarlo() 37](#_Toc31329052)

[Game Class 37](#_Toc31329053)

[Game.\_\_init\_\_() 37](#_Toc31329054)

[Game.handle\_input() 38](#_Toc31329055)

[Game.endgame() 39](#_Toc31329056)

[Game.save\_game() 40](#_Toc31329057)

[Game.clear\_save() 40](#_Toc31329058)

[Game.run() 40](#_Toc31329059)

[Menu Class 42](#_Toc31329060)

[Menu.\_\_init\_\_() 42](#_Toc31329061)

[Menu.display() 42](#_Toc31329062)

[Menu.user\_choose() 42](#_Toc31329063)

[Menu.instructions() 43](#_Toc31329064)

[instructions.txt 43](#_Toc31329065)

[Menu.new\_game() 45](#_Toc31329066)

[Menu.load\_game() 45](#_Toc31329067)

[Menu.run() 46](#_Toc31329068)

[1: Menus 56](#_Toc31329069)

[2: Instructions 56](#_Toc31329070)

[3: Loading/Saving 56](#_Toc31329071)

[4: Gameplay – Local Multiplayer Mode 56](#_Toc31329072)

[5: Gameplay – AI Mode 56](#_Toc31329073)

[6: AI 57](#_Toc31329074)

[Types of Invalid Moves: 57](#_Toc31329075)

[Testing Video 65](#_Toc31329076)

[Objective Analysis 66](#_Toc31329077)

[End-User Feedback: 67](#_Toc31329078)

[How easy is the application to use? 67](#_Toc31329079)

[Does the application fulfil the objectives, as shown in the table? 67](#_Toc31329080)

[Any criticisms? 67](#_Toc31329081)

[Any improvements or extensions? 67](#_Toc31329082)

[Analysis of end-user feedback 67](#_Toc31329083)

[Possible extensions 68](#_Toc31329084)

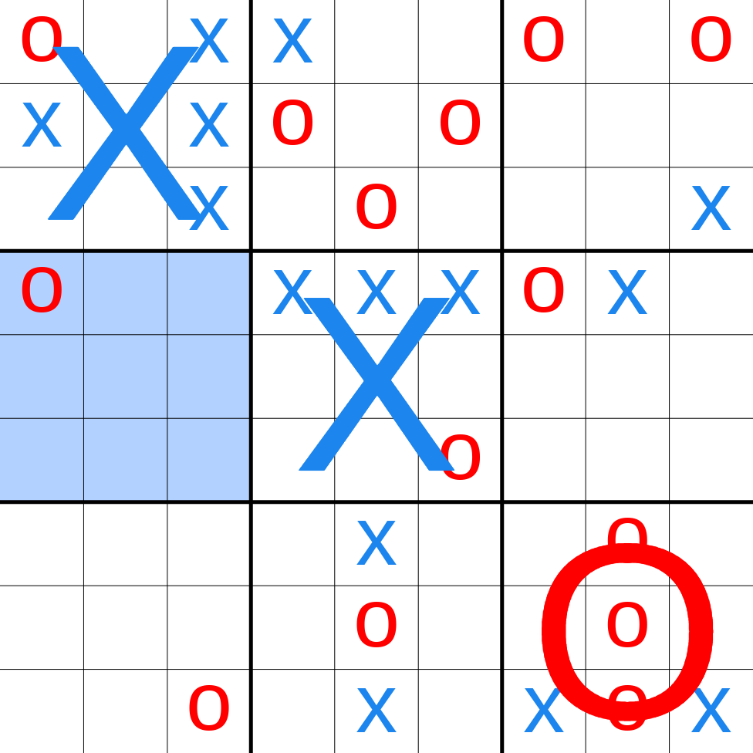
Analysis

# Introduction:

In this project, I will be attempting to create a game of Ultimate Tic Tac Toe playable with an AI mode at different levels of difficulty, as well as with a local multiplayer mode.

Ultimate Tic Tac Toe is a variation of the more popular ‘Tic Tac toe’ and is played in a similar way, except there is an added layer of strategy and difficulty.

Definitions:



‘Local grid’ refers to the smaller grids, one of which is shown in a dashed border.

‘Global grid’ refers to the entire grid, shown in a solid border.

The player ‘symbols’: ‘X’ or ‘O’ are placed where the corresponding player has chosen.

Ultimate Tic Tac Toe consists of a 3 by 3 global Tic Tac Toe grid containing local Tic Tac Toe grids.

The first player can position their symbol anywhere on the global grid.

The position of the first player’s symbol on the local grid, corresponds to the position of the local grid on the global grid the next player is able to position their symbol in.

For example, if the first player chooses the top right position in the central local grid, the second player must position their symbol in an empty position in the top right local grid.

Local grids that have been won are marked for that player.

If a player is directed to a local grid that is full (has no empty spaces left), or that has already been won/lost, the player may place their symbol anywhere on the global grid.

For example, if the first player chooses the centre position on the top local grid, the second player is directed to the central local grid, which is full, so the second player may position their symbol on any empty space on the global grid.

The objective of the game is to win 3 connected local grids before the opponent, similar to in Tic Tac Toe.

Ultimate Tic Tac Toe is a difficult game to grasp and play strategically, hence it is difficult to find players near your skill level. Furthermore, Ultimate Tic Tac Toe is not very famous because of its higher complexity and the inability to play it recreationally on paper as with Tic Tac Toe.

An AI would be the perfect solution to this problem since it would mean that any user is able to play with the AI opponent on the difficulty level that best suits them, giving a more difficult and fun gameplay experience. I propose to counter the unpopularity of the game by boosting enthusiasm for it via competitive local multiplayer games.

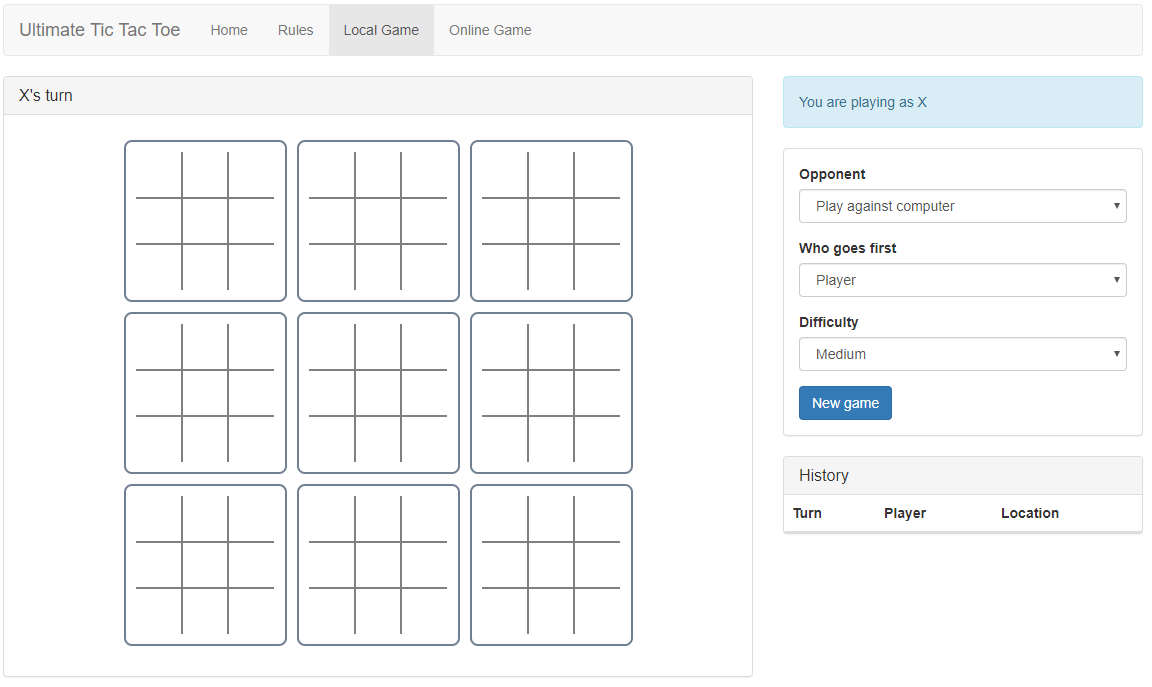
# Background Research:

## Competing Products:

In my research I have found two existing Ultimate tic Tac toe games that incorporate AI.

Game 1: [Ultimate Tic Tac Toe](https://ultimate-t3.herokuapp.com/), Game 2: [Strategic Tic Tac Toe](https://www.coolmathgames.com/0-strategic-tic-tac-toe)

Overall Functionality and Appearance:

Although both games incorporate AI, Ultimate Tic Tac Toe includes 8 levels of difficulty, while Strategic Tic Tac Toe has only 1 mode of difficulty. Both games allow Local Multiplayer as well as AI.

Strategic Tic Tac Toe includes a flashy look with many colours and sound effects when pressing buttons or making moves, while Ultimate Tic Tac Toe is more streamlined, clean, and functionality driven.

Since Ultimate Tic Tac Toe is more simplistic in appearance, has more difficult modes of AI and has a system for friendly online play, it seems to be directed towards adults and teenagers. While on the other hand, Strategic Tic Tac Toe’s aesthetic, design and lack of functionality suggests it is directed to children.

Since the audience of my game are teenagers and adults, I will be borrowing more heavily from the functionality-driven design of Ultimate Tic Tac Toe as opposed to Strategic Tic Tac Toe.

### Key Features:

Ultimate Tic Tac Toe is a difficult game to grasp and interpret, so both games have developed similar features so the user understands what is happening at every stage in the gameplay:

* Both systems have a clear menu, so that the user can select what modes they would like to play in.
* Both games have a tutorial page, which explains the games lesser known and complex rules clearly.
* In Strategic Tic Tac Toe, a position must be selected twice as a confirmation, to ensure it is not selected by accident, and on the first selection, the complimentary local grid the opponent will have to make their move in is highlighted to give the user reminder of the effect of their move.
* Both games highlight the local grid/s the current player is allowed to play in so the user is sure of what they can and cannot do.
* Ultimate Tic Tac Toe includes a history bar, which shows all the previous moves that have taken place.

I will attempt to include the first two features to ensure any confusion is dispelled so that the users can focus on playing the game. The history bar feature, however, seems unnecessary to me since the entire history of the game would be saved anyway as symbols on the global grid.

One feature that I did not find in either game which I think is useful is highlighting the last move, so the current player knows what happened last turn. This feature would be enough to replace both the history bar, and the local grid highlighting, because the previous move would clearly show which local grids the current player is allowed to play in.

This feature is particularly useful in the late game, when the board is filled with symbols, many of which might compliment the current local board, so finding the last move may be difficult.

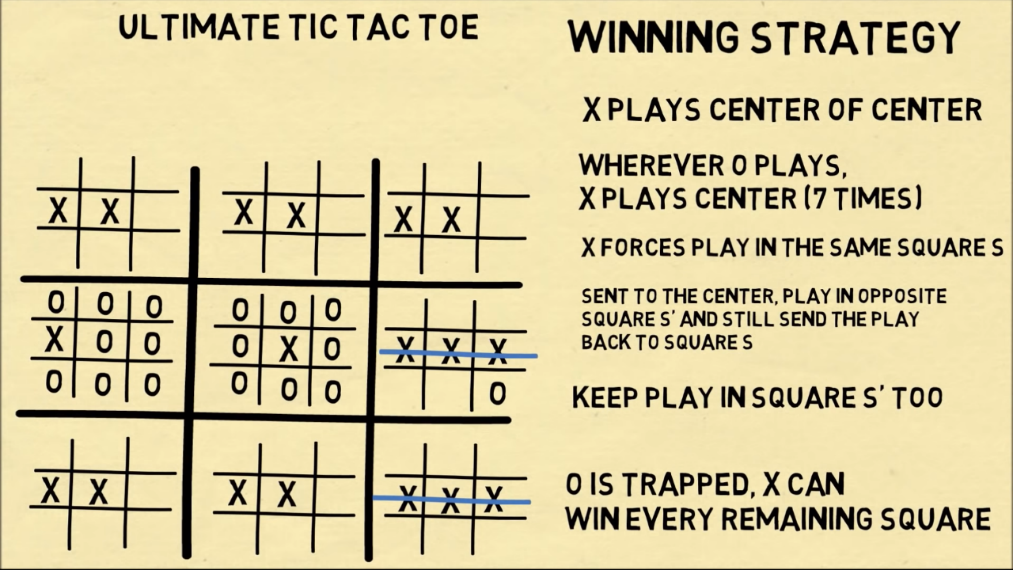
### Rule Variation:

In the games of Tic Tac Toe which I have researched, there is a difference in approach on what happens when a player is directed, by the opposition, to a local grid that is already full or has already been won (and so is replaced by the winner’s symbol).

In the minority of games, when a player is directed to a local grid that has been won but is not yet completely full, the player is forced to play in that local grid. If the player is directed to a local grid that is full, the player is then allowed to place their symbol in any local grid.

In majority of games, however, when a player is directed to a local grid that has been won (regardless of it is full or not), or one that is full, the player may place their symbol on any local grid.

In my research, I have found that there is a cheat strategy to win the game if the minority rule is followed:



1. Player 1 starts by playing in the centre of the centre local grid.
2. Player 2 is forced to play away from the centre in the centre local grid.
3. Player 1 then continues to choose the centre in all local grids it is directed to.
4. This way, Player 1 has secured the centre position, which is the strongest position, in almost all the local grids.
5. This means Player 1 is almost guaranteed to win the game, since Player 2 is trapped between possible victories for player 1.

This strategy would destroy the fun of the game since the first player would always win, making it tedious to play. For this reason, my game will use the majority rule in which players that are directed to a grid which has already been won (or one that is full) can place their symbol anywhere on the global grid.

## AI:

### Issues and Complexity

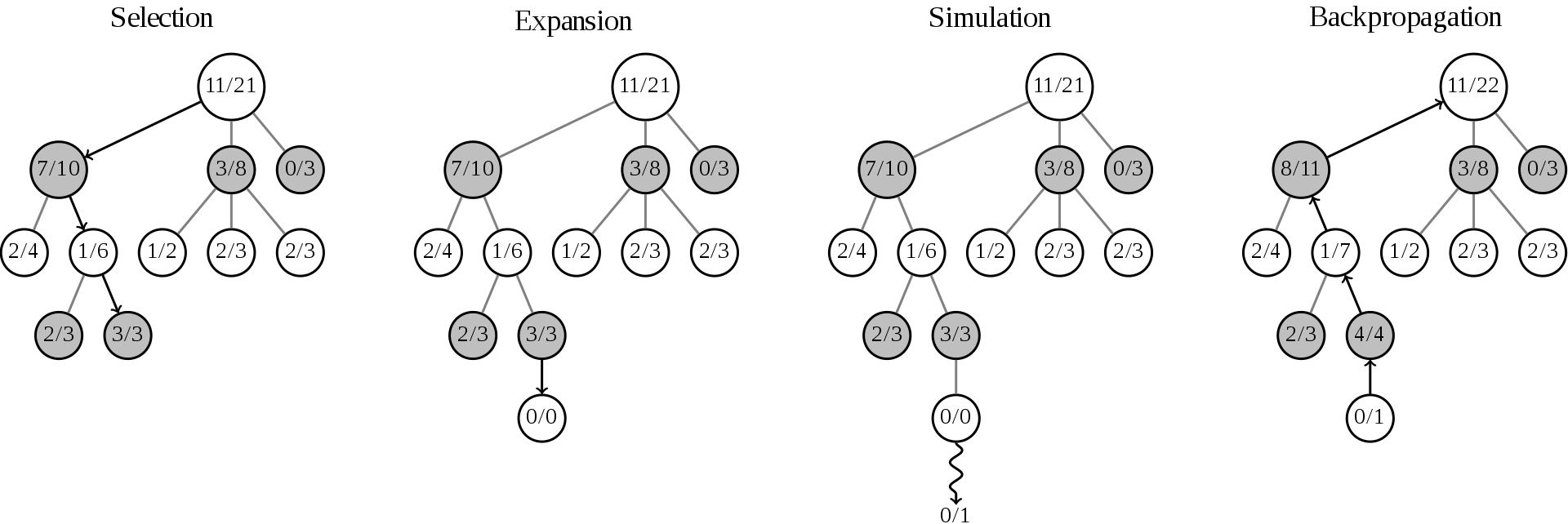
Ultimate Tic Tac Toe is significantly more complex than other variations of Tic Tac Toe.

The difficulty of creating a good AI for Ultimate Tic Tac Toe is in the difficulty of balancing the wins of local boards with the more significant winning of the global board. It is further difficult to anticipate the moves of the opponent, and to react accordingly, since a massive variety of different strategies can be applied.

The hardship of planning ahead, the balancing act of knowing if the local board or global board is more significant to consider in a particular move, as well as the fact that a move that is seen as bad in one turn may later be recalled as very good, all make it very challenging to create a good AI for Ultimate Tic Tac Toe.

### Monte Carlo Tree Search Algorithm:

For the AI of my game, it will be necessary to create a tree called a game tree to store the legal game states which could occur in the game, and an algorithm will be required to evaluate the possible game states and give a result on which game states are best, and so giving a result on which move the AI should make. I have chosen to use the Monte Carlo Tree Search Algorithm for my project.



* In the diagram seen above, there are 4 directed graphs, or in this case, game trees.
* Each node on each game tree represents a legal game state.
* The grey and white nodes are game states caused by moves made by the two different players. White representing player 1, black representing player 2.
* The node at the top of the game tree is the root node, which represents the current game state.
* The nodes following the root node connected to the root node are all child nodes, except the bottom nodes which are called leaf nodes.
* All nodes except the root node represent possible legal game states, and the edges connected the nodes represent moves the AI or opponent can make.
* Each node has a fraction value associated with it, the numerator is the number of wins the AI has obtained from simulating games from its child nodes, while the denominator is the total number of games that have been simulated from its child nodes.

The Monte Carlo Tree Search Algorithm follows 4 steps:

1. Selection:

This is where the AI starts from the root node, and continues to select a child node, based on certain selection criteria (explained in more detail in ‘AI 🡪 UCT equation’), until a leaf node is reached. The leaf nodes are nodes which have not yet been fully expanded.

1. Expansion

Unless the leaf node ends the game with a win/loss or draw, a child node of the leaf node is produced, which represents a game state that is the result of a legal move that can be made from the leaf node.

1. Simulation:

A simulation is made from the game state of the child node, where random moves are selected until a win/loss or draw is reached.

1. Back propagation:

If the result of the game is a loss, the child’s node value becomes 0/1, if the result of the simulated game is a win, the child’s node value becomes 1/1 and if it is a draw, the value becomes 0.5/1. This value is then added to the values of all the parent nodes associated with it. In the diagram, the result of the simulation is 0/1 (loss), so the numerator and denominator are added to the fraction on each of the child node’s white parent nodes (only white is added since the simulation being considered is for the white player.)

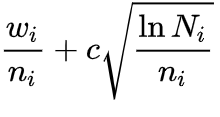
This process continues until either a set amount of time has been reached, or until a given number of iterations have been reached.

After stopping, the child node of the root (2nd row of nodes after the root node) which has the highest denominator in its node value is chosen as the next move.

### UCT Equation:

The selection step in the Monte Carlo Tree Search algorithm requires certain selection criteria to tell it which parts of the game tree to explore next.

In most Monte Carlo Tree Search algorithms, a variation of the following UCT formula is used as the selection criteria.

This equation seeks to balance ‘exploitation’ with ‘exploration’ which is the basis of the Monte Carlo Tree Search algorithm.

‘Exploitation’ will cause the algorithm to select leaf nodes of parts of the tree that have the highest number of wins compared to simulations.

While ‘Exploration’ will cause the algorithm to select leaf nodes from parts of the tree that have the smallest number of simulations.

Wi/ni represents the exploitation part of the equation, while the rest of it represents the exploration part of the equation.

Exploitation allows for better results for the path that seems to lead to the highest number of wins, to ensure that it does actually lead to the highest number of wins, while exploration ensures that any nodes that are better or that have been overlooked are not ignored. When these 2 factors are balanced, the path with the highest win rate is most likely to be selected.

In the selection step, the algorithm calculates the UCT formula on all the nodes from the child nodes of the root node, all the way to the leaf nodes, and selects the leaf node that maximises this value.

Wi/ni makes up the fraction value of the node currently being considered, Wi being the number of wins that node, or its children have achieved and ni being the number of simulations that have occurred on the node or on its children.

C is a constant that varies depending on the specific project being done, and is chosen by testing different values and seeing which obtains the greatest results. It is the value that signifies the importance of exploration relative to exploitation or vice versa.

Ni is the total number of simulations that have been run on the parent node of the one considered, i.e. the number of times the parent node of the node considered has been selected and simulated.

The UCT formula allows the algorithm to work at great efficiency, making sure to select the correct parts of the game tree to explore next to ensure that the direct child nodes of the root node are most accurate.

Once the algorithm has been stopped, the direct child node of the root node (2nd row of nodes) who has been visited the most, will be selected by the AI as the best move. This is because this node has most often been selected by the UCT function since it has the highest proportion of wins while also having a low proportion of unexplored future moves associated with it, meaning its win rate is accurate.

### Monte Carlo Tree Search- Advantages & Disadvantages:

In my research, I had found 2 main algorithms which can be used for Ultimate Tic Tac Toe: Minimax and Monte Carlo Tree Search.

I believe Monte Carlo Tree Search is more suited than Minimax for this project for the following reasons:

* No complex heuristic evaluation function is required to check how good a particular game state is. This is crucial since Ultimate Tic Tac Toe lacks a simple heuristic evaluation function i.e. it is difficult to check whether a game state is good or not. This is because there is a very subjective and difficult balancing act between local grids and the global grid, and because of the extreme complexity of the game.

On the other hand, Minimax relies completely on a good heuristic evaluation function, which makes it implausible for me to use.

* Majority of the games of Ultimate Tic Tac Toe that I have researched make use of Monte Carlo Tree Search as opposed to Minimax, and the few that use Minimax are generally massively outperformed by Monte Carlo Tree Search AI.
* Monte Carlo Tree Search would generally be faster since Minimax must explore a large amount of possibilities of future game states and perform calculations on them, which in this game, is somewhat less than 81 factorial game states (somewhat less due to alpha-beta pruning).

Monte Carlo Tree Search on the other hand, instead of relying on searching through most of the possibilities in the entire game, makes use of large samples of random simulations which can occur much more quickly, and it does not need to search through the entire game tree: only the most beneficial parts of the game tree are selected to be explored further (using the UCT equation).

* Another reason Monte Carlo Tree Search would be faster is because it can be stopped at any time or after a given number of iterations, and still produce an answer. If only a small number of iterations have been performed, this answer would not be very accurate so a large number of iterations is required for a good answer, but on the other hand Minimax must perform a lot of searching and calculations and complete them all, it cannot be stopped half way.

# Data Flow Diagrams and Objectives:

## Main Game:

### 1-Main Menu:

Once the application is launched, the user will be met with a main menu.

Here, they will be greeted with a clear, simplistic design and a welcome message, and will be given options regarding what they would like to do next.

This menu must include:

1. A brief, introductory, inviting, welcome message
2. A ‘quit’ option that allows the user to exit the application, and stop it running.
3. A ‘Tutorial’ option that takes the player to a tutorial page.
4. A ‘New Game’ option, which takes the player to the play game mode options page.
5. A ‘Load Game’ option which loads the last saved game, or displays a message saying there is no saved game.
6. If time allows, a GUI should be created for the menu as well as the rest of the game.
7. The menu should be simplistic and clear, the inputs required to select a certain option must be clear, as well as the look of the options themselves.
8. All Invalid inputs should be handled with a message saying the input is invalid, and by asking the user for the input again.

### 2-Tutorial Page:

Users who have never played Ultimate Tic Tac Toe before will benefit from the tutorial page.

It will be a comprehensive, yet concise explanation of how the game works, and allow the player to navigate to other pages.

The tutorial must include:

1. A brief but concise explanation of the rules of the game and how to win.
2. Should explain the game with an example.
3. Should explain which moves are invalid and which are valid.
4. Should be organised in an easy to read way.
5. Should be able to navigate the tutorial using any input.
6. Should have a ‘back’ button to allow the player to go back to the main menu page.

### 3-Play Game Menu:

Due to the many different game modes in this project, a game options page is required.

This must include:

1. Options for ‘AI Play’, and ‘Local Multiplayer Play’
2. Each option should be clear, and how to select it should be made clear.
3. The ‘AI Play’ option should take the user to the ‘AI difficulty menu’
4. The ‘Local Multiplayer Play’ option should start a local multiplayer game if chosen.
5. All Invalid inputs should be handled with a message saying the input is invalid, and by asking the user for the input again.

### 4-Change AI Difficulty Menu:

After selecting the ‘AI Play’ game mode, this menu should appear which asks the user to select an AI difficulty level.

This must include:

1. There must be at least 3 modes of difficulty available, or 4 if time allows.
2. If a difficulty mode is selected, a game in the mode Human vs AI should start with the AI set to the difficulty chosen.
3. There must be a back button which takes the user back to the main menu page.
4. All Invalid inputs should be handled with a message saying the input is invalid, and by asking the user for the input again.

### 5-Initialise Game:

Once a game mode is selected, the game will be initialised.

1. The data structure storing the current game state should be created.
2. The data structure storing the previous move should be created.
3. An instance of the Monte Carlo class should be created if in human vs AI mode.
4. There must be a 50% chance for either player (Human or AI, Player 1 or Player 2) to go first and have the first move.

### 6-AI:

1. There must be 3 or 4 modes of difficulty to start the AI with, with each mode being significantly more difficult than the previous.
2. The AI must make its move, and fulfil its turn in a reasonable amount of time for all difficulties.
3. The AI must be just as good as other online AI products.

### 7-Human vs AI mode:

1. A turn-based system showing correctly which player’s turn it is – Human vs AI
2. The correct winning player must be shown at the end of the game.

### 8-Local Multiplayer Mode:

The is the mode of play where 2 players play against each other on the same device.

This must include:

1. A turn-based system showing correctly which player’s turn it is – Player 1 or Player 2.
2. The correct winning player must be shown at the end of the game.

### 9-Display Game:

Before and after every move, the game board as well as some game information should be shown.

This must include:

1. The data structure storing the current game state information must be used to display the current global game grid, with all symbols on the grid present in their correct places.
2. The last move made should be clearly shown.
3. The boundaries of the local grid (or grids) that the current player is allowed to play in must be clear to prevent confusion.
4. The name of the current player (player 1/player 2/Human Player/AI Player) must be shown.

### 10-User Make Move/Input:

The user is given an intuitive and simple method to input their symbol onto the global grid, where desired. This input is then checked to see if it is a number between 11 and 99 inclusive to see if the coordinate could fit on the global grid.

This must include:

1. The user must be able to enter an option to quit the current game and save, and return back to the main menu.
2. The user must be asked for the coordinate of the global grid where they would like to input their symbol.

The user should be asked to input two numbers, from 1-9, where the location of each number on the keypad on the keyboard, compliments the area being referred to.

The first number entered must compliment the local grid the user would like to place their symbol in, while the second number compliments the exact place the user would like to place their symbol.

For example, 99 would be the top right of the global grid, or 59 would be the top right of the middle global grid.

1. Once the input has been received, it must be validated to see if it is in the syntax described above. This can be done by checking that the length of the input is 2, and that both of the numbers in the input are between 1 and 9 inclusive.
2. Once a valid move is input, it must change the data structure of the current game state accordingly.
3. A ‘back’ input option must be available at every point in the game, allowing the player to save and exit the current game and move back to the main menu.

### AI Analyse and Make Move:

The AI must perform the Monte Carlo Tree search algorithm on the game tree and return the best possible valid move based on its current difficulty setting.

The specific objectives are mentioned under ‘AI data flow diagram and objectives’.

### 11-Validate Movement:

Once a valid input is entered, the move must be checked and validated to see if it is correct within the rules of the game.

This must be done by:

1. Checking that a symbol is not already present in the area that is specified.
2. Check that the area specified is not a full (won/lost/drawn) local grid.
3. Checking that the area specified is in the local grid the previous player’s move corresponds to.
4. If any of these checks return True, the user is given an ‘invalid move’ message, and asked for their move input again.

### 12-Check Win:

Each local grid and the global grid from the current game state data is checked for a win.

This must:

1. Check for a win on both the local grids and global grid effectively: checking vertical, horizontal and all diagonals to check if there are any three-in-a-rows.
2. If a win is spotted on a local grid, that part of the current global grid data is marked with the symbol of the player that has won it.
3. If the global grid has been won/drawn/lost the end screen should be displayed.
4. The global grid must be checked after every move to see if the global grid or local grids have been won/lost/drawn.

### 13-End Screen:

Shows who won, or shows a draw, and takes the user back to the main menu.

Once the game has been completed with a result, the end screen is shown.

This must include:

1. A clear indicator of if the user has won, lost or drawn the game.
2. The user is taken back to the main menu, where they can select to play a new game if they wish.

### 14-Save Game/Load Game:

The game must be saved at every point the global grid is not empty, and load correctly.

1. The factors making up the state of the current game: the previous move, current player, global grid, game mode, and AI difficulty (if applicable) are all saved.
2. The game should not be saved if the game has been won/lost/drawn or no move has yet been made
3. The game is saved after every turn in gameplay.
4. The game is saved when a user presses the back button in game.
5. When the ‘Load game’ option is chosen in the main menu, the correct last save is run, so that the loaded game is identical to the original unfinished game.
6. When ‘Load game’ option is chosen in the main menu, and no game is save is available, then display a message saying no game save is available.
7. Any new game saves should overwrite the old game save.

## AI Data Flow Diagram and Objectives:

### 15-Initialise Game Tree

1. The data structure that stores the nodes must be initialised within the root node, the children of which are the child nodes the children of which are their child nodes, and so on.

### 16-Select Node

1. The UCT value must be calculated for a node at every depth from the root node using information stored in the game tree.
2. The node that has the highest UCT value must be continually selected from the children of the node last selected (or from the root node), until a leaf node is reached.

### 17-Expand

Once a leaf node is selected, a child node of it must be produced.

1. A child node of the leaf node must be produced by randomly making a legal move from the game state of the leaf node.

### 18-Simulate

This child node must then be simulated to find the (win, simulation) value for it.

1. From the game state of the child node, random legal moves are made until an end state is reached: loss, win or draw.
2. If the end state is a loss, the (win, simulation) value for the game state becomes (0, 1), if the end state is a win, the value becomes (0, 1), and if the end state is a draw, the value must become (0.5, 1).

### 19-Back-Propagate

Once the value of the child node is calculated, this is used to update the rest of its parent nodes.

1. The numerator and denominator of the (win, simulation) value for the child node must then be added to its parent, and then to the parent’s parent, and so on, until it is added to the root node.

### 20-Make Move

1. The selection, expansion, simulation, and back propagation steps must be repeated until a given number of iterations run out.
2. Once the iterations run out, the child node of the root node which has the highest number of simulations (the highest denominator in its (win, simulation) value) must be chosen.
3. The AI must then choose the move conveyed by the chosen node, and update the game tree data structure accordingly.

### 21-Discard Game Tree

1. Once the move has been made by the AI, the game tree data structure must then be discarded and emptied and a new one should be used for the next AI move.

### 22-C Constant Selection

1. The C constant of the UCT value should be selected carefully, to ensure a balance between exploration and exploitation so that the AI works well.

# Modelling

## Monte Carlo Algorithm Pseudocode Model

Function Select(Node): #Returns the leaf node with the highest UCT value

While True:

if Node.children:

for child\_node in Node.children:

if child\_node is UCTmaximiser:

Node = child\_node

Else:

Return Node

Function Expand(Node, GameTree): # adds Node children to GameTree, returns random childnode

GameTree.add\_nodes(Node.possible\_children)

Return random.select(Node.possible\_children)

Function Simulate(Node): # Simulates random game starting with node, returns outcome of game

While Node.possible\_children:

Node = Random.select(Node.possible\_children)

If Node IS win:

Node.value = (1, 1)

If Node IS loss:

Node.value = (0, 1)

If Node IS draw:

Node.value = (0, 1)

Function BackPropogate(Node): # Adds the node’s values to all of its parents

For Parent in Node.parents:

Parent.Value = Parent.Value + Node.Value

ss

# Main Program:

Iterations = 0

While iterations <= Specified Iterations:

Selected\_leaf = Select(Game\_Tree.root)

If selected\_leaf.number\_of\_simulations == 0:

Simulation\_Node = Selected\_leaf

Else:

Simulation\_Node = Expand(Selected\_leaf, Game\_Tree)

Simulate(Simulation\_Node)

BackPropogate(Simulation\_Node)

Iterations = iterations + 1

For move\_node in GameTree.root.children:

If move\_node.number\_of\_simulations IS Highest:

AI.make\_move(move\_node)

These functions demonstrate the outline of what must be produced in each of the functions when coding the project using python and the main program shows how the different functions should be put together to produce a result for the AI.

## Monte Carlo Algorithm Python Model

The following is a python model, using classes, of the working AI algorithm on a normal tic tac toe game.

Currently, this model is only able to produce the first move that the AI would make, I will be building on this for my project.

1. **import** math
2. **import** random
4. game\_tree = {0: []}

7. **class** Node:
8. **def** \_\_init\_\_(self, parent, children, state=None, UCT=None, root=False):
9. self.parent = parent
10. self.children = children
11. self.value = (0, 0)
12. self.state = state
13. self.root = root
14. **if** root:
15. self.depth = 0
16. **else**:
17. self.depth = self.parent.depth + 1
18. self.add\_to\_game\_tree()
20. **def** add\_to\_game\_tree(self):
21. **if** **not** self.root:
22. **if** **not** len(game\_tree) > self.depth:
23. game\_tree[self.parent.depth + 1] = []
24. game\_tree[self.depth].append(self)
26. **def** \_\_repr\_\_(self):
27. **return** '{}, {}, {}'.format(self.state, self.value, self.depth)
29. **def** display\_node(self):
30. **for** row **in** self.state:
31. **for** pos **in** row:
32. **if** pos == ' ':
33. **print**('☐', end='')
34. **else**:
35. **print**(pos, end='')
36. **print**()
37. **print**(self)
39. **def** print\_lineage(self):
40. node = self
41. **print**(node)
42. **while** node.parent != None:
43. **print**(node.parent)
44. node = node.parent
45. **else**:
46. **print**(node)

49. game\_state = [[' ', ' ', ' '],
50. [' ', ' ', ' '],
51. [' ', ' ', ' ']]
53. results = []
55. root = Node(parent=None, children=[], state=game\_state, root=True)

58. **class** MonteCarlo:
59. **def** \_\_init\_\_(self, grid, turn=1):
60. self.local\_grid = grid
61. self.symbols = ['X', 'O']
62. self.player = 1
63. self.C = 1
64. self.turn = turn
66. **def** get\_UCT(self, node):
67. **if** node.parent != None:
68. W = node.value[0]
69. n = node.value[1]
70. N = node.parent.value[1]
71. **if** n == 0:
72. **return** math.inf
73. **else**:
74. **return** W/n + (self.C \* math.sqrt(math.log(N)/n))
75. **else**:
76. **return** math.inf
78. **def** select(self, node):
79. UCT\_maximum = 0
80. UCT\_maximiser = None
81. **if** node.children:
82. **for** child\_node **in** node.children:
83. **if** self.get\_UCT(child\_node) > UCT\_maximum:
84. UCT\_maximum = self.get\_UCT(child\_node)
85. UCT\_maximiser = child\_node
86. **return** UCT\_maximiser
87. **else**:
88. **return** node
90. **def** check\_move(self, grid, coordinate):
91. x, y = coordinate
92. **if** grid[y][x] != ' ':
93. **return** False
94. **else**:
95. **return** True
97. **def** get\_children(self, node):
98. **for** x **in** range(3):
99. **for** y **in** range(3):
100. **if** self.check\_move(node.state, (x, y)):
101. new\_child = Node(node, [])
102. new\_child.state = [x[:] **for** x **in** node.state]
103. new\_child.state[y][x] = self.symbols[self.turn % 2]
104. node.children.append(new\_child)
106. **def** expand(self, node):
107. self.get\_children(node)
108. **return** random.choice(node.children)
110. **def** simulate(self, selected\_node):
111. self.get\_children(selected\_node)
112. copy\_node = selected\_node
113. **while** len(copy\_node.children) > 0:
114. copy\_node = random.choice(copy\_node.children)
115. self.get\_children(copy\_node)
116. W1, n1 = selected\_node.value
117. W2, n2 = self.check\_win(copy\_node.state)
118. selected\_node.value = (W1 + W2, n1 + n2)
120. **def** back\_propagate(self, simulated\_node):
121. **while** simulated\_node.parent != None:
122. W1, n1 = simulated\_node.parent.value
123. W2, n2 = simulated\_node.value
124. simulated\_node.parent.value = (W1 + W2, n1 + n2)
125. simulated\_node = simulated\_node.parent
127. **def** Monte\_Carlo(self):
128. **while** self.turn <= 100:
129. selected\_leaf = self.select(game\_tree[0][0])
130. **if** selected\_leaf.value[1] == 0:
131. self.simulate(selected\_leaf)  # VALUE IS ADDED TO SELECTED LEAF
132. self.back\_propagate(selected\_leaf)
133. **else**:
134. simulation\_node = self.expand(selected\_leaf)
135. self.simulate(simulation\_node)
136. self.back\_propagate(simulation\_node)
137. self.turn += 1
138. self.make\_move().display\_node()
140. **def** make\_move(self):
141. simulation\_max = 0
142. move\_node = None
143. **for** node **in** game\_tree[1]:
144. **if** node.value[1] > simulation\_max:
145. simulation\_max = node.value[1]
146. move\_node = node
147. **return** move\_node
149. **def** get\_winners(self, grid):
150. winners = []
151. # horizontal
152. **for** x **in** range(len(grid)):
153. winners.append(grid[x])
155. # vertical
156. **for** y **in** range(len(grid[0])):
157. col = []
158. **for** row **in** range(len(grid)):
159. col.append(grid[row][y])
160. winners.append(col)
162. right\_down = []
163. left\_down = []
165. **for** y **in** range(len(grid)):
166. **for** x **in** range(len(grid[y])):
167. **if** y == x:
168. right\_down.append(grid[y][x])
169. **if** y == -x + 2:
170. left\_down.append(grid[y][x])
171. winners.append(right\_down)
172. winners.append(left\_down)
173. **return** winners
175. **def** board\_filled(self, grid):
176. **for** row **in** grid:
177. **if** ' ' **in** row:
178. **return** False
179. **return** True
181. **def** check\_win(self, grid):
182. **for** row **in** self.get\_winners(grid):
183. **if** row == [self.symbols[self.player - 1]] \* 3:
184. **return** 1, 1  # WIN
185. **elif** row == [self.symbols[self.player - 2]] \* 3:
186. **return** 0, 1  # LOSS
188. **if** self.board\_filled(grid):
189. **return** 0.5, 1  # DRAW

192. mont = MonteCarlo(game\_state)
194. mont.Monte\_Carlo()

Output varies from time to time, but generally the algorithm tends to choose a corner position or the centre position. This would suggest that the model is successful since these positions do tend to be the strongest.

Example Output: has chosen bottom left corner

☐☐☐

☐☐☐

O☐☐

[[' ', ' ', ' '], [' ', ' ', ' '], ['O', ' ', ' ']], (14, 20), 1

Design

My project is an AI mode and Local Multiplayer Mode for the game Ultimate Tic Tac Toe, in which the ‘Monte Carlo Tree Search’ algorithm is used by the AI player to make its moves.

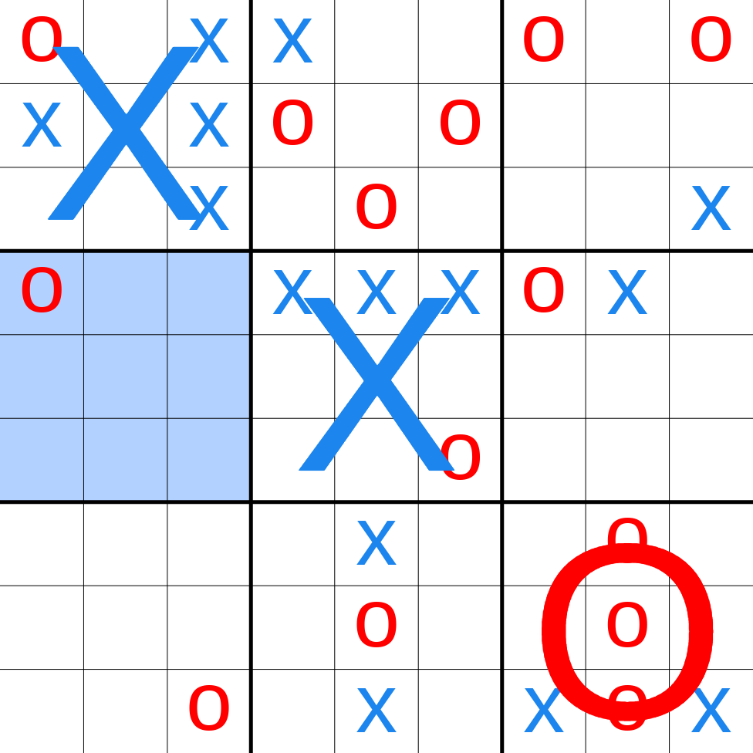
# Project Technology

I will be using the python language (version 3.7.0) as well as the Pycharm Integrated Development Environment to produce the Technical Solution implementation of my project.

I will be using the python built-in math library to calculate values for my Monte Carlo Tree Search algorithm, and the built-in random library to choose random moves, Nodes, etc. wherever necessary.

# Game Rules:

Definitions:



‘Local grid’ refers to the smaller grids, one of which is shown in a dashed border.

‘Global grid’ refers to the entire grid, shown in a solid border.

The player ‘symbols’: ‘X’ or ‘O’ are placed where the corresponding player has chosen.

Ultimate Tic Tac Toe consists of a 3 by 3 global Tic Tac Toe grid containing local Tic Tac Toe grids.

The first player can position their symbol anywhere on the global grid.

The position of the first player’s symbol on the local grid, corresponds to the position of the local grid on the global grid the next player is able to position their symbol in.

For example, if the first player chooses the top right position in the central local grid, the second player must position their symbol in an empty position in the top right local grid.

Local grids that have been won are marked for that player.

If a player is directed to a local grid that is full (has no empty spaces left), or that has already been won/lost, the player may place their symbol anywhere on the global grid.

For example, if the first player chooses the centre position on the top local grid, the second player is directed to the central local grid, which is full, so the second player may position their symbol on any empty space on the global grid.

The objective of the game is to win 3 connected local grids (diagonal, vertical, or horizontal) before the opponent, similar to in Tic Tac Toe.

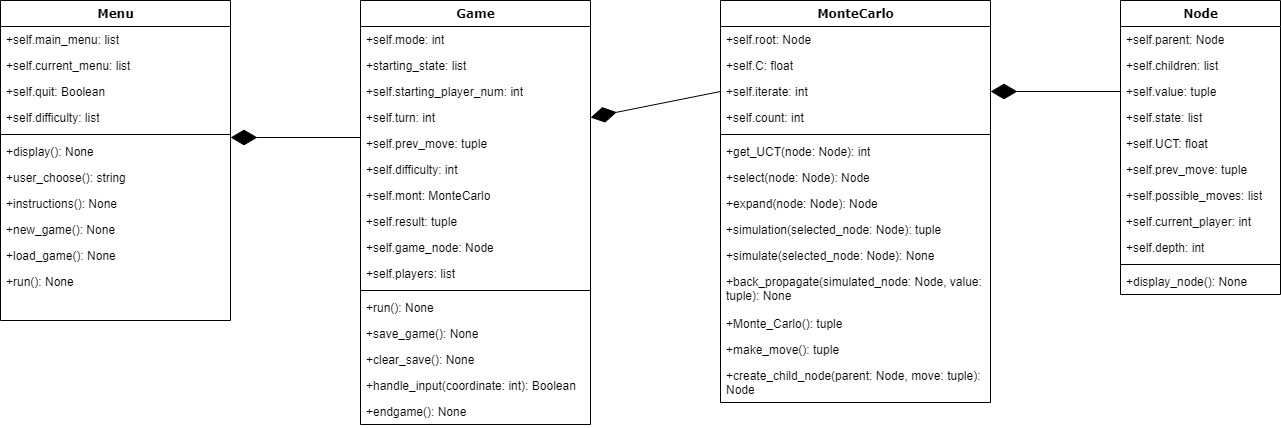
# File Structure and Organisation

In the same directory as the python program file, 2 text files are present.

The first is the ‘tutorial.txt’ file which must be downloaded alongside the python program file for it to work. It is used to print the tutorial pages, further explained in ref.

The second is the ‘save\_file.txt’, this is generated by the python program and is used to save the game and load saved games. Further explained in ref.

# Class Diagram

The following is a high-level overview of the details and relationships of the classes in my project.

## Classes

### Node Class:

The Node class is used to represent different game states or nodes, and stores the variables associated with them.

The associated variables and methods are explained further in ‘Key Data Structures’ and ‘Key Functions 🡪 Node Class’.

### MonteCarlo Class:

The main algorithm in this project is the Monte Carlo Tree Search algorithm, which is carried out and managed by the MonteCarlo class.

See algorithm details in Analysis 🡪 Monte Carlo Tree Search Algorithm.

The associated variables and methods are explained further in ‘MonteCarlo Class’

### Game Class:

The Game class runs the Ultimate Tic Tac Toe game, and stores an instance of the MonteCarlo class, making the AI move using MonteCarlo.MonteCarlo().

It includes methods for a turn-based game system, input handling, and win/loss/draw handling, all of which and its variables are further explained in ‘Game Class’.

### Menu Class

The Menu class controls the various menus and navigation options to quit the application, go to the instructions page, run a local multiplayer game, and run human vs AI games. Variables and methods are explained in ‘Menu Class’.

# Key Data Structures:

Explain key data structures in greater detail. Why they are used.

## Node.state

Stores the current game state (global grid) in a list containing 9 strings of length 9.

The entire list is equivalent to the global grid.

The 9 strings correspond to the 9 local grids.

The 9 characters in each string correspond to the values in each local grid.

* Each local grid consists of 9 positions, which are represented by the corresponding symbols in the string.
* The global grid is read from top left to bottom right, and the local grids are read from top left to bottom right.
* If no symbol is present at that position in the grid, it is substituted by an empty space in that position in the string.

For example, for the image in ‘Game Rules’, the Node.state value would be:

[‘O XX X X’, ‘X O O O ’, ‘O O X’, ‘O ’, ‘XXX O’, ‘OX ’, ‘ O’, ‘ X O X ’, ‘ O O XOX’]

‘Game state’ and ‘global grid’ are used interchangeably and refer to a list representing the global grid in the above format, within or outside of the Node class.

## Node.parent

Refers to the ‘parent’ Node instance of the current Node instance, such that:

Node.parent + legal move = Node

Node.parent is a Node that precedes Node by 1 legal move.

Node can be produced by Node.parent by performing the legal move.

## Node.children

A list containing all ‘children’ of the Node, which are instances of the Node class, such that:

Node + legal move = child\_node

For every child\_node in Node.children, child\_node.parent is Node.

## Node.value

Node.value is a tuple containing 2 integers: W, N

N is the number of times the node has been used to simulate a full game of Ultimate Tic Tac Toe by the MonteCarlo class.

If a simulation is won, 1 is added to W, and if a simulation is drawn, 0.5 is added to W, if it is lost, nothing is added to W, in all cases 1 is added to N.

## Node.prev\_move and Node.current\_player

Node.current\_player is an integer, 1 or 0, resembling whether the next move will be an ‘O’, or an ‘X’.

Node.prev\_move is a tuple of length 2, it contains coordinates of the move that has been carried out on Node.parent.state to give Node.state (not in keypad format).

Where Node.parent.state =

[‘O XX X X’, ‘X O O O ’, ‘O O X’, ‘O ’, ‘XXX O’, ‘OX ’, ‘ O’, ‘ X O X ’, ‘ O O XOX’]

And:

Node.prev\_move = (3, 1)

Node.current\_player = 1

Node.state =

[‘O XX X X’, ‘X O O O ’, ‘O O X’, ‘O**O** ’, ‘XXX O’, ‘OX ’, ‘ O’, ‘ X O X ’, ‘ O O XOX’]

At the 3rd grid, on the 1st position, an O replaces the previously empty space.

## Node.possible\_moves

Stores all the valid moves that can be played on the Node instance.

The valid moves are generated by calling the get\_valid\_moves function, which is further explained in Key Functions → get\_valid\_moves.

This keeps track of all the moves required to ‘expand’, or produce, all of the valid child Nodes from the Node.

## Node.depth

Node.depth keeps track of the depth the node is at in the game tree, this is equivalent to the number of moves that have taken place between root Node (the initial/original Node) and Node.

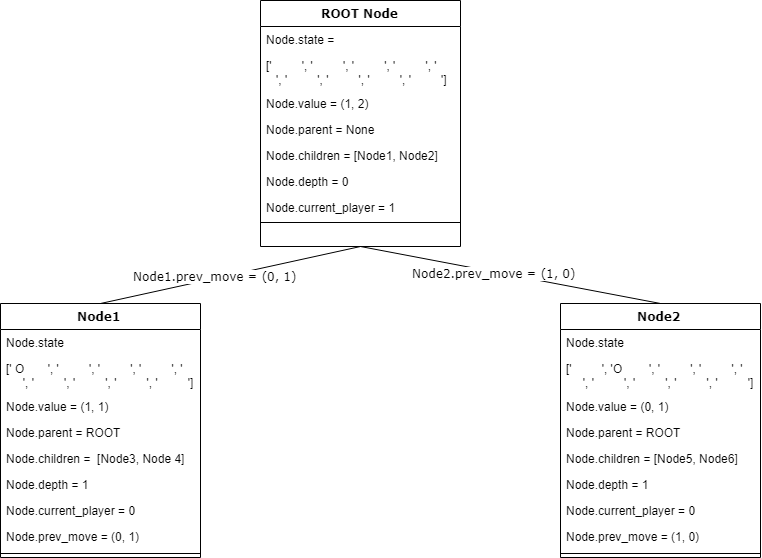
At the root Node, Node.depth = 0, children of the root have Node.depth = 1, children of these have Node.depth = 2 and so on.

By default, Node.depth = Node.parent.depth + 1 and root node.depth = 0.

## Node.UCT

The Monte Carlo Tree Search Class calculates the UCT value for each node using the MonteCarlo.get\_UCT() method. The UCT value calculated for a Node is stored in its Node.UCT.

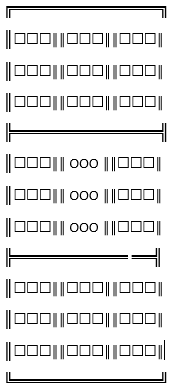
## Game Tree:

The Nodes, their parents and children, form a game tree, which looks like this.

Note: ROOT.prev\_move = None. Not all variables are shown.

# Game I/O

## Node Display



During gameplay the Node.state of a particular Node can be displayed in the format shown on the left using the method Node.display\_node().

It has the following advantageous characteristics:

* The 9 local grids are clearly separated by borders.
* All empty spaces are replaced with a ‘☐’ symbol to ensure they can be clearly seen.
* All won/lost grids are replaced by 9 symbols instead of one large

symbol, as in other games, to further increase clarity of which local grids have been won or lost.

* The global grid is surrounded by a border, to further emphasize the contents of the global grid and make them clear.

## Node.display\_node()

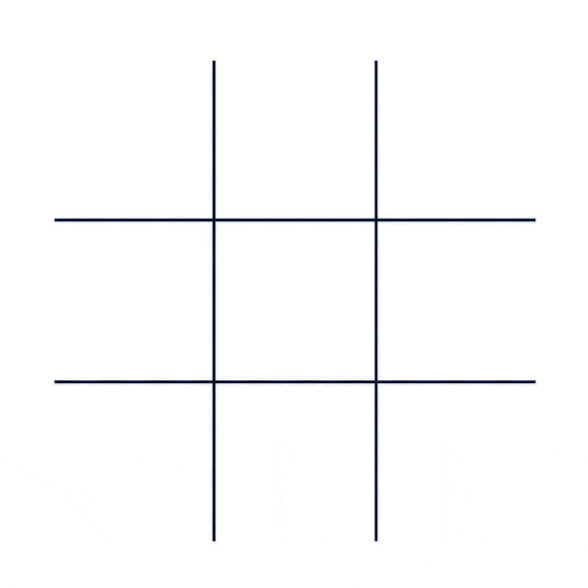
Where a local grid has been detected as being won by ‘O’ or ‘X’, it is replaced by a single ‘O’ or ‘X’ in the Node.state. This local grid containing a single ‘O’ or ‘X’ is then temporarily replaced by a local grid full of ‘O’s or ‘X’s in order to clearly mark that that local grid has been won by that player.

def display\_node(self):  
 *# Parameters :- None  
 # Return Type :- None  
 # Purpose :- Prints the Node.state in a clear format* state\_copy = self.state[:]  
 for grid in range(len(state\_copy)):  
 if state\_copy[grid] == 'O':  
 state\_copy[grid] = 'OOOOOOOOO'  
 elif state\_copy[grid] == 'X':  
 state\_copy[grid] = 'XXXXXXXXX'  
  
 print('╔' + '═' \* 16 + '╗')  
 for z in range(3):  
 for i in range(3):  
 for grid in state\_copy[z \* 3:(z + 1) \* 3]:  
 print('║' + grid[i \* 3:(i + 1) \* 3].replace(' ', '#'), end='║')  
 print()  
 if z == 2:  
 print('╚' + '═' \* 16 + '╝')  
 else:  
 print('╠' + '═' \* 16 + '╣')  
 print()

## Move Input

When the user is prompted on their turn for a move, it is entered in the syntax below, using the keypad on the keyboard:

**Ultimate Tic Tac Toe Local Grids:**



1

2

3

4

5

6

7

8

9

The position of the numbers on the keypad corresponds to the position of the local grid on the global grid, and also to the position of the desired empty space in the local grid.

First a local grid is chosen by selecting the corresponding number on the keypad.

Then a position on that local grid is chosen by selecting the corresponding number on the keypad.

So, to place the X on the top right of the global grid, the input must be ‘99’.

This input method allows moves to be made very quickly and accurately, and is easy to understand.

After the user inputs the move, the keypad tuple is converted into a tuple which refers to the intended place on the global grid in the input\_convertor() function.

For the keypad string of ‘99’ the new tuple would be (2, 2).

### Function: input\_convertor()

This is a global function which converts the keypad input of 2 characters into a tuple referring to a position in the global grid list.

def input\_convertor(coordinate, reverse=False):

# Parameters :- coordinate:string, reverse:boolean

# Return Type :- tuple

# Purpose :- Converts string 2 digit coordinate into keypad tuple or vice versa

###########################

conv = [7, 8, 9, 4, 5, 6, 1, 2, 3]

x, y = int(coordinate[0]), int(coordinate[1])

if not reverse:

acc\_coordinate = (conv.index(x), conv.index(y))

else:

acc\_coordinate = (conv[x], conv[y])

return acc\_coordinate

Process:

* reverse=False would convert the tuple into the keypad format tuple.
* ‘75’ is input as the coordinate, in keypad format, this refers to the central position on the top left local grid.
* The list conv is initialized.
* x is set to the integer 7, and y is set to the integer 5.
* The index that 7 and 5 are present at in conv are made into a tuple, and initialized as acc\_coordinate, which is then returned.

# Key Functions

## Node Class:

|  |  |  |  |
| --- | --- | --- | --- |
| Method Name | Parameters | Return Types | Purpose |
| \_\_init\_\_ | state: list,  parent: Node instance, children: list,  value: tuple,  prev\_move: tuple, current\_player: integer,  depth: integer,  possible\_moves: list,  self.UCT = None  self.root = Boolean | None | Initialises all the variables associated with the Node instance.  Default values for root Node:   * value = (0, 0) * current\_player = 1 * depth = 0   Default values for non-root Nodes:   * value = (0, 0) * current\_player = opposite of Node.parent.current\_player * depth = Node.parent.depth + 1   Further details in ‘Key Data Structures’. |
| display\_node | None | None | Prints the Node.state in the format demonstrated in Game UI → Node Display.  Further info can be found here. |

## Global Functions:

|  |  |  |  |
| --- | --- | --- | --- |
| Function Name | Parameters | Return Types | Purpose |
| check\_move | grid: list  coordinate: tuple | Boolean | Grid refers to the Node.state of a particular Node.  Coordinate refers to a tuple that is the move that is being considered.  This function checks if a particular move is valid.  Further explained in Key Functions → Function: check\_move |
| get\_valid\_moves | Node\_state: list  Prev\_move: tuple | List | Finds all the valid child nodes of a Node, so that Node + valid move = child node.  It then returns them as a list.  Further explained in Key Functions → Function: get\_valid\_moves |
| check\_win | global\_grid: list | Tuple | global grid is the Node.state of a particular Node.  This function checks if each local grid in the global grid, and the global grid itself, has ended, and whether the game has been won/lost or drawn.  Further explained in Key Functions → Function check\_win |

## Function: opposite()

This function gives the opposite of the input integer, returning 1 for 0 and 0 for 1.

SUBROUTINE

opposite(dig):  
 if dig == 0:

return 1

else:

return 0

## Function check\_win()

Checks if a terminal state (win, loss, or draw) is reached in any of the local grids, and then checks if a terminal state is reached in the global grid.

If a local grid has reached a win or loss, the string corresponding to it in the Node.state is replaced by a string of length 1 of the winning symbol, this makes the won/lost local grids very easy to distinguish.

Node.display\_node() checks if each local grid contains a single ‘O’ or ‘X’ and if it does, it replaces the single symbol by 9 symbols in a local copy of the global grid, so that the global grid contains only 1 of the symbol, but 9 are displayed to show that the local grid is full.

For example, if the first local grid has been won by the ‘O’ player, it will be replaced by a single ‘O’ in the Node.state. When Node.display\_node() is run, a display\_list is produced which is a copy of the Node.state, and contains 9 symbols at the won local grid instead of 1.

def check\_win(global\_grid):  
 for i in range(len(global\_grid)):  
 current\_grid = global\_grid[i]  
 if len(current\_grid) > 1:  
 for winner in winners:  
 a, b, c = winner  
 grid\_winner = current\_grid[a] + current\_grid[b] + current\_grid[c]  
 if grid\_winner == 'XXX':  
 global\_grid[i] = 'X'  
 break  
 elif grid\_winner == 'OOO':  
 global\_grid[i] = 'O'  
 break  
  
 for winner in winners:  
 a, b, c = winner  
 global\_winner = global\_grid[a] + global\_grid[b] + global\_grid[c]  
 # try:  
 # global\_winner = global\_grid[a] + global\_grid[b] + global\_grid[c]  
 # except IndexError:  
 # pass  
 if global\_winner == 'OOO':  
 return 1, 1  
 elif global\_winner == 'XXX':  
 return 0, 1  
  
 for grid in global\_grid:  
 if ' ' in grid:  
 return None  
 return 0.5, 1

## Function: check\_move()

Takes 2 arguments, grid and coordinate.

Grid refers to the Node.state of a particular Node.

Coordinate refers to a tuple and is the move that is being considered (non-keypad format).

This function determines whether or not a tuple move is valid by checking that the corresponding position on the global grid is empty, and does not contain a single ‘O’ or ‘X’.

SUBROUTINE

check\_move(grid, coordinate):  
 a, b ← coordinate  
 IF LEN(grid[a]) ≤ 1:  
 return FALSE

if grid[a][b] ≠ ' ':  
 return FALSE  
  
 return TRUE

ENDSUBROUTINE

## Function: get\_valid\_moves

SUBROUTINE

get\_valid\_moves(node\_state, prev\_move):  
 valid\_moves = []  
 if prev\_move IS NONE:  
 for a in range(9):  
 for b in range(9):  
 valid\_moves.append((a, b))  
  
 else:  
 x, y = prev\_move  
 if ' ' not in node\_state[y]:  
 for a in range(9):  
 for b in range(9):  
 if check\_move(node\_state, (a, b)):  
 valid\_moves.append((a, b))  
 else:  
 for a in range(9):  
 if check\_move(node\_state, (y, a)):  
 valid\_moves.append((y, a))  
  
 return valid\_moves

ENDSUBROUTINE

# Monte Carlo Class

The main algorithm in this project is the Monte Carlo Tree Search algorithm, which is carried out and managed by the MonteCarlo class.

See algorithm details in Analysis 🡪 Monte Carlo Tree Search Algorithm.

## MonteCarlo.\_\_init\_\_(self)

Initialises the variables in the Monte Carlo Tree Search class (constructor function).

|  |  |
| --- | --- |
| Parameter | Purpose |
| self.grid | The current game state (or global grid) in the gameplay, in the same syntax as in Key Data Structures 🡪 Node.state |
| prev\_move | The previous move that has taken place in the game, same as Key Data Structures 🡪 Node.prev\_move |
| self.root | An instance of the Node class.  It is the root node in the game tree, and so it is initialised with the following required variables:  parent=None, children=[], state=self.grid, prev\_move=prev\_move, depth=0  More information can be found in:  Key Functions 🡪 Node Class 🡪 Init  The self.root.UCT is set to Infinity. |
| self.iterate | The number of iterations that the Monte Carlo Tree Search algorithm performs before making its move. |
| self.count | Stores the number of iterations that have been performed. |
| self.C | Stores the C value for the Monte Carlo Tree Search Algorithm.  Usually chosen empirically, most commonly equal to 2\*\*0.5. |

## MonteCarlo.create\_child\_node

parent is an instance of Node

move is a move in the format Key Data Structures 🡪 Node.prev\_move

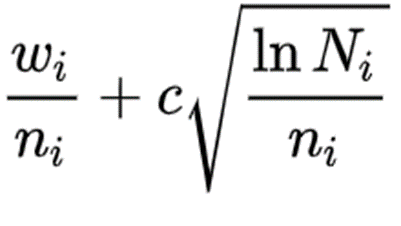
Creates and returns the child node of a parent, such that Node + move = Child Node

def create\_child\_node(self, parent, move):  
 child\_node = Node(parent, [], prev\_move=move, state=parent.state[:])  
   
 a, b = move  
 child\_node.state[a] = child\_node.state[a][:b] + symbols[parent.current\_player] + child\_node.state[a][b + 1:]  
   
 parent.possible\_moves.remove(move)  
 child\_node.parent.children.append(child\_node)  
 return child\_node

## MonteCarlo.get\_UCT()

Uses the UCT formula to calculate and return the UCT value of a Node.

Further information in ‘Analysis 🡪 AI 🡪 UCT Equation’.

def get\_UCT(self, node):  
 if node.parent:  
 W = node.value[0] # Number of wins of node and all its children in simulations  
 n = node.value[1] # Number of visits to node  
 N = node.parent.value[1] # Number of visits to node.parent  
 if n == 0:  
 return math.inf  
 else:  
 return W / n + (self.C \* math.sqrt(math.log(N) / n))  
 else:  
 return math.inf

see ref for more information

## MonteCarlo.select()

Finds a leaf node (a node that has not been fully expanded) using the path that maximises the UCT equation. Further information in ‘Analysis 🡪 AI 🡪Monte Carlo Tree Search Algorithm’.

def select(self, node):  
 while len(node.possible\_moves) == 0:  
 for child\_node in node.children:  
 child\_node.UCT = self.get\_UCT(child\_node)  
  
 children\_sorted = sorted(node.children, reverse=True, key=lambda each\_node: each\_node.UCT)  
 node = children\_sorted[0]  
  
 # Randomise equal nodes:  
 equal\_UCT\_nodes = []  
  
 for sorted\_node in children\_sorted:  
 if sorted\_node.UCT == node.UCT:  
 equal\_UCT\_nodes.append(sorted\_node)  
  
 if len(equal\_UCT\_nodes) > 0:  
 node = random.choice(equal\_UCT\_nodes)  
 return node

## MonteCarlo.simulation()

Simulates a game from a selected node using random valid moves, until a terminal state (win/loss/draw) is reached. It then returns the value of the terminal state as in ‘Key Functions 🡪 Function: check\_win’.

Further information in ‘Analysis 🡪 AI 🡪Monte Carlo Tree Search Algorithm’.

def simulation(self, selected\_node):  
 current\_player = selected\_node.current\_player  
 current\_state = selected\_node.state[:]  
 prev\_move = selected\_node.prev\_move  
  
 is\_terminal = check\_win(current\_state)  
  
 while is\_terminal is None:  
 possible\_moves = get\_valid\_moves(current\_state, prev\_move)  
  
 random\_move = random.choice(possible\_moves)  
  
 ra, rb = random\_move  
 current\_state[ra] = current\_state[ra][:rb] + symbols[current\_player] \  
 + current\_state[ra][rb + 1:]  
 is\_terminal = check\_win(current\_state)  
 prev\_move = random\_move  
 current\_player = opposite(current\_player)  
  
 return is\_terminal

## MonteCarlo.simulate()

Simulates a game from a node by calling the simulation function, and then updates the node.value for that node. Finally, it calls the back\_propogate function to update all the parents of the node aswell as the root node using the result of the simulation.

Further information in ‘Analysis 🡪 AI 🡪Monte Carlo Tree Search Algorithm’.

def simulate(self, selected\_node):  
 sim\_result = self.simulation(selected\_node)  
  
 W1, n1 = selected\_node.value  
 W2, n2 = sim\_result  
 selected\_node.value = (W1 + W2, n1 + n2)  
 self.back\_propagate(selected\_node, (W2, n2))

## MonteCarlo.expand()

Expands a Node, by creating a new child node for it, and then returns the new child node.

Further information in ‘Analysis 🡪 AI 🡪Monte Carlo Tree Search Algorithm’.

def expand(self, node):  
 expansion\_move = random.choice(node.possible\_moves)  
 new\_child\_node = self.create\_child\_node(parent=node, move=expansion\_move)  
  
 return new\_child\_node

## MonteCarlo.back\_propogate()

Updates all the parent Nodes of a Node until the root is reached, using the terminal value from the Node simulation. Further information in ‘Analysis 🡪 AI 🡪Monte Carlo Tree Search Algorithm’.

def back\_propagate(self, simulated\_node, value):  
 W2, n2 = value  
 while simulated\_node.parent is not None:  
 W1, n1 = simulated\_node.parent.value  
 simulated\_node.parent.value = (W1 + W2, n1 + n2)  
 simulated\_node = simulated\_node.parent

## MonteCarlo.make\_move()

Finds the node with the highest number of visits (simulations) from the immediate children of the root node (MonteCarlo.root) at depth=1, in accordance with the Monte Carlo Tree Search algorithm.

Further information in ‘Analysis 🡪 AI 🡪Monte Carlo Tree Search Algorithm’.

def make\_move(self):  
 max\_visits = 0  
 move\_node = None  
  
 for node in self.root.children:  
 if node.value[1] >= max\_visits:  
 max\_visits = node.value[1]  
 move\_node = node  
  
 return move\_node

## MonteCarlo.MonteCarlo()

Runs the Monte Carlo Tree Search algorithm by calling the various methods of the Monte Carlo class.

Further information in ‘Analysis 🡪 AI 🡪Monte Carlo Tree Search Algorithm’.

def Monte\_Carlo(self):  
 while self.count <= self.iterate:  
 # select the node with the highest UCT value  
 selected\_leaf = self.select(self.root)  
  
 # if Node hasn't been visited  
 if selected\_leaf.value[1] == 0:  
 # simulate a game from the Node and back propagate it  
 self.simulate(selected\_leaf)  
  
 else: # if Node has been visited  
 # expand the Node  
 simulation\_node = self.expand(selected\_leaf)  
   
 # and simulate the new Child Node  
 self.simulate(simulation\_node)  
  
 self.count += 1  
   
 # AI move is the immediate child node   
 # of the root that has been visited most  
 move\_node = self.make\_move()

# Game Class

The Game class runs the Ultimate Tic Tac Toe game, and stores an instance of the MonteCarlo class, making the AI moves using MonteCarlo.MonteCarlo().

It includes functions for a turn-based game system, input handling, and win/loss/draw handling.

## Game.\_\_init\_\_()

Initialises the variables in the Game class.

|  |  |
| --- | --- |
| Game Class Variable | Purpose |
| self.mode=int(mode) | This is the mode the game will be running in: self.mode = 1: Human vs AI mode, or self.mode = 0: local multiplayer mode (Player 1 vs Player 2). |
| difficulty | This is the number of iterations the AI will be running at. This is set by default to None, and is kept as None where the game mode is local multiplayer, since AI will not be used. |
| starting\_state | The initial state of the game to be run, if a starting state is not entered, the default value is an empty global grid. |
| self.starting\_player\_num=random.choice([0, 1]) | Stores the value of the player which will take the first turn in the game.  By default, this is set to a random choice so that:  50% of time: Player 1/Human 50% of time: Human/AI |
| self.turn=self.starting\_player\_num | Stores the value of the player which is yet to move in the current turn, and is updated as the game progresses.  Equals self.starting\_player\_num to ensure the player stored in self.starting\_player\_num is the starting player. |
| self.prev\_move=None | Stores the move that has been chosen in the previous turn by the AI or Human in the format seen in ref. |
| self.mont = MonteCarlo(iterate=self.difficulty, grid=starting\_state, prev\_move=self.prev\_move) | Stores the instance of the Monte Carlo Class that will be used to make the AI moves. |
| self.result=None | Stores the result of the game.  None=The game has not yet completed  (1, 1) = Game has been won  (0.5, 1) = Game has been drawn  (0, 1) = Game has been lost |
| self.game\_node=self.mont.root | Stores the current root game tree node. |
| self.players = [['Player 1', 'Player 2'], ['Human', 'AI']] | Stores the possible players so that,  self.players[self.mode][self.turn] refers to the correct current player’s turn. |

## Game.handle\_input()

Takes the parameter coordinate the valid form of which is a string comprising of 2 digits corresponding to where the human player would like to place their symbol on their turn. The 2 digit string directly to the corresponding positions on the global grid, they are not in keypad format.

Returns True if the coordinate is valid, and False if the coordinate is invalid.

def handle\_input(self, coordinate):

# Parameters :- coordinate:tuple

# Return Type :- Boolean

# Purpose :- check if the input coordinate is valid or not

if coordinate == '000':

return None

if len(coordinate) != 2:

return False

for val in coordinate:

if val not in ['1', '2', '3', '4', '5', '6', '7', '8', '9']:

return False

if int(coordinate) > 100 or int(coordinate) < 1:

return False

a, b = input\_convertor(coordinate)

move\_check = check\_move(self.mont.root.state, (a, b))

if move\_check is False:

return move\_check

if self.prev\_move is not None:

if len(self.game\_node.state[self.game\_node.prev\_move[1]]) != 1 \

and a != self.game\_node.prev\_move[1]:

return False

* 000 is the quit game string, so if the coordinate is the quit string, it is ignored so the game can be quit later in the code.
* If the length of the string is greater than 2, it cannot be a coordinate, since it must have 2 characters to refer to the global grid.
* If there are any non-number characters used, then the coordinate is invalid.
* The coordinate is then turned into a tuple which directly references the desired global grid position.
* The new coordinate is then checked by the check\_move function to see if it is valid
* Lastly the new coordinate is checked to see that it refers to a location in the local grid corresponding to the previous move except if the previous move refers to a local grid that is full.

## Game.endgame()

Checks the Game.result and outputs the relevant message on screen.

def endgame(self):

# Parameters :- None

# Return Type :- None

# Purpose :- Displays a Win/Loss/Draw string at end of game

if self.result == (1, 1):

print("{} Won!".format(self.players[self.mode][opposite(self.turn)]))

elif self.result == (0, 1):

print("{} Won!".format(self.players[self.mode][opposite(self.turn)]))

else:

print("Draw!")

* Opposite(self.turn) is used instead of self.turn since during the Game.run() method, the opposite() function is used one more time after the game ends, and so it no longer refers to the player that has made the last move i.e. the player that has won.

## Game.save\_game()

Saves the current game state into a text file called ‘save\_file.txt’ so that an identical game can be loaded with all the same variables. Such that the player is then able to pick up where they left off with a previous unfinished game.

def save\_game(self):

save\_file = open('save\_file.txt', 'w')

for local\_grid in self.game\_node.state:

save\_file.write(local\_grid + '\n')

save\_file.write(str(self.turn) + '\n')

save\_file.write(str(self.prev\_move[0]) + str(self.prev\_move[1]) + '\n')

save\_file.write(str(self.difficulty) + '\n')

save\_file.write(str(self.mode))

save\_file.close()

The global grid, self.turn, self.prev\_move, self.difficulty, and self.mode are all saved.

## Game.clear\_save()

def clear\_save(self):

open('save\_file.txt', 'w').close()

This erases the current save of the game.

This method is called when a game has ended- there is no need to save a game that has already ended.

## Game.run()

The game system for Ultimate Tic Tac Toe.

def run(self):

# Parameters :- None

# Return Type :- None

# Purpose :- Returns the Ultimate Tic Tac Toe Game

self.game\_node.display\_node()

if self.prev\_move is not None:

print('previous:', input\_convertor(self.prev\_move, reverse=True))

while self.result is None:

if self.prev\_move is not None:

self.save\_game()

if self.turn == 1 and self.mode == 1:

print("AI Turn")

if self.game\_node.state == default\_starting\_state:

self.game\_node = self.mont.create\_child\_node(self.mont.root, (4, 4))

self.prev\_move = (4, 4)

else:

self.game\_node = self.mont.Monte\_Carlo()

self.prev\_move = self.game\_node.prev\_move

elif self.turn == 0 or self.mode == 0:

print(self.players[self.mode][self.turn] + ' Turn')

human\_move = input('Coordinates: ')

if human\_move == '000':

print('Game state saved and game aborted')

return None

while self.handle\_input(human\_move) is False:

print("Invalid Move!")

human\_move = input('Coordinates: ')

if human\_move == '000':

print('Game state saved and game aborted')

return None

a, b = input\_convertor(human\_move, reverse=False)

self.game\_node.state[a] = self.game\_node.state[a][:b] + symbols[self.turn] + self.game\_node.state[a][b + 1:]

self.prev\_move = (a, b)

self.game\_node.prev\_move = (a, b)

self.result = check\_win(self.game\_node.state)

self.game\_node.display\_node()

print('previous:', input\_convertor(self.prev\_move, reverse=True))

if self.mode == 1:

self.mont.\_\_init\_\_(grid=self.game\_node.state, iterate=self.difficulty, prev\_move=self.prev\_move)

self.turn = opposite(self.turn)

self.endgame()

self.clear\_save()

* At the start of the game, the global grid and (if it is not None i.e. if the game has been loaded) self.prev\_move is displayed.
* This ensures that a new game starts by displaying an empty global grid, and that a loaded game starts by displaying the saved global grid, and the last turn before saving.
* If self.prev\_move is not None checks that that a move has been made before the game saves since there is no point in saving a game where no move has yet been made- an empty global grid.
* If self.turn == 1 and self.mode == 1 checks that the mode is human vs AI and the turn is 1, meaning turn 1 is the AI’s turn if the mode is human vs AI.
* The AI move is defaulted to (4, 4), the central position, if no move has taken place on the global grid, as this is the best move.
* Elif self.turn == 0 or self.mode == 0 is true where the mode is local multiplayer or if the mode is human vs AI but the turn is 0.
* The human’s chosen move is then checked, and if it is 000, the game is immediately aborted, the game has already been saved before this move, and a message is displayed.
* The human’s move is then checked to see if it is invalid, and is given an error message until the move is either valid or 000.
* The human move string is then converted into a tuple which refers to the global grid by the input\_convertor function.
* The global grid is then changed to add the correct symbol to the chosen position in the global grid, and self.prev\_move and self.game\_node.prev\_move are updated.
* Then self.result is run, to check if the game has ended, and to replace any won/lost local grids with a single ‘O’ or ‘X’. The single ‘O’s and ‘X’s are later displayed as local grids full of the ‘O’ or ‘X’ to mark them as won/lost.
* Once the game has ended, the self.endgame() is called and the save\_file.txt is cleared.

# Menu Class

The Menu class handles all menus and runs all the different game options, it stores Game class instances.

## Menu.\_\_init\_\_()

Constructor method for Menu, initialises all Menu variables.

|  |  |
| --- | --- |
| Menu Class Variable | Purpose |
| self.main\_menu = ['New Game', 'Load Game', 'Instructions', 'Exit'] | Stores the options for the main menu. |
| self.current\_menu = self.main\_menu | Stores the value for the current main menu – the menu being currently displayed and handled. |
| self.quit = False | If True, the Menu.run() stops. |
| self.difficulties = [100, 1000, 5000] | Contains the number of iterations for the 3 difficulties of AI, self.difficulties[0] = iterations for easy AI and so on. |

## Menu.display()

def display(self):

for i in range(len(self.current\_menu)):

print(i + 1, '-', self.current\_menu[i])

Displays the current menu in the following format:

1 – option 1

2 – option 2

The user would then input ‘1’ to choose option 1 and so on.

## Menu.user\_choose()

def user\_choose(self):

option = input(': ')

while option not in [str(x) for x in range(1, len(self.current\_menu) + 1)]:

print('Invalid')

option = input(': ')

return option

Gets the user choice for the current menu, if the user choice is not in options, a message saying invalid is printed, and user is asked for their option again until a valid choice is input.

## Menu.instructions()

def instructions(self):

instructions\_file = open('instructions.txt', 'r', encoding='utf8')

instruction\_lines = instructions\_file.readlines()

for line in instruction\_lines:

if 'Page' in line:

if input(':') == '0':

print()

break

print(line, end='')

print()

instructions\_file.close()

Prints every line in the instructions.txt file, if ‘page’ is in the line, the user is asked for an input so that they have a chance to read the previous page before they move on to the next.

If the user inputs ‘0’, the method breaks.

## instructions.txt

The contents of ‘instructions.txt’ which are printed in the instructions page are as follows:

Ultimate Tic Tac Toe Instructions\nPress ENTER to continue\nType \'0\' to go back\n  
Page 1 of 7:  
When in game, input '000' to save and abort the game, returning you back to the main menu.  
Ultimate Tic Tac Toe consists of a 3 by 3 global Tic Tac Toe grid containing local Tic Tac Toe grids:  
╔════════════════╗  
║###║║###║║###║  
║###║║###║║###║  
║###║║###║║###║  
╠════════════════╣  
║###║║###║║###║  
║###║║###║║###║  
║###║║###║║###║  
╠════════════════╣  
║###║║###║║###║  
║###║║###║║###║  
║###║║###║║###║  
╚════════════════╝  
  
Page 2 of 7:  
The first player can position their symbol anywhere on the global grid using the keypad:  
The location of numbers on the keypad correspond to the locations on the global grid.  
To place the symbol on the centre of the top-middle local grid:  
  
-Input the number at the top of the keypad (9), this selects the local grid  
-Then select the number at the centre of the keypad (5), which selects the location at that local grid.  
╔════════════════╗  
║###║║###║║###║  
║###║║#X#║║###║  
║###║║###║║###║  
╠════════════════╣  
║###║║###║║###║  
║###║║###║║###║  
║###║║###║║###║  
╠════════════════╣  
║###║║###║║###║  
║###║║###║║###║  
║###║║###║║###║  
╚════════════════╝  
The previous move is shown after every turn:  
previous: 85  
  
Page 3 of 7:  
The second player can then choose from the local grid, corresponding to the location chosen by the first player.  
  
Here, the second player must choose a location from the bottom right local grid, corresponding to the first player's choice.  
╔════════════════╗  
║###║║###║║###║  
║###║║#X#║║###║  
║###║║###║║###║  
╠════════════════╣  
║###║║###║║###║  
║###║║#O#║║###║  
║###║║###║║###║  
╠════════════════╣  
║###║║###║║###║  
║###║║###║║###║  
║###║║###║║###║  
╚════════════════╝  
previous 55  
  
Page 4 of 7:  
If a player wins a local grid, it is marked for them  
╔════════════════╗  
║###║║###║║###║  
║###║║###║║###║  
║###║║###║║###║  
╠════════════════╣  
║###║║OOO║║###║  
║###║║OOO║║###║  
║###║║OOO║║###║  
╠════════════════╣  
║###║║###║║###║  
║###║║###║║###║  
║###║║###║║###║  
╚════════════════╝  
  
Page 5 of 7:  
To win, you must win 3 local grids in a row.  
╔════════════════╗  
║###║║###║║OOO║  
║###║║###║║OOO║  
║###║║###║║OOO║  
╠════════════════╣  
║###║║OOO║║###║  
║###║║OOO║║###║  
║###║║OOO║║###║  
╠════════════════╣  
║OOO║║###║║###║  
║OOO║║###║║###║  
║OOO║║###║║###║  
╚════════════════╝  
Computer Won!  
  
Page 6 of 7:  
╔════════════════╗  
║###║║###║║OOO║  
║###║║#X#║║OOO║  
║###║║###║║OOO║  
╠════════════════╣  
║###║║###║║###║  
║###║║###║║###║  
║###║║###║║###║  
╠════════════════╣  
║###║║###║║###║  
║###║║###║║###║  
║###║║###║║###║  
╚════════════════╝  
previous 99  
  
In this case, the local grid the second player has been directed to by the first player is already full,  
so the second player may place their symbol anywhere on the global grid.  
  
Page 7 of 7  
╔════════════════╗  
║###║║###║║OOO║  
║###║║#X#║║OOO║  
║###║║###║║OOO║  
╠════════════════╣  
║###║║###║║###║  
║###║║###║║###║  
║###║║###║║###║  
╠════════════════╣  
║###║║###║║###║  
║###║║##O║║###║  
║###║║###║║###║  
╚════════════════╝  
previous 26

This is an elaborate explanation of: the game rules, how to input moves, the valid moves, and the invalid moves, how to win the game, how the game and previous move are displayed, as well as other things.

Everything is explained with examples to make it clear, and in bite-sized, understandable chunks.

## Menu.new\_game()

def new\_game(self):

self.current\_menu = ['Human vs AI', 'Local Multiplayer']

self.display()

mode\_choice = self.user\_choose()

s if mode\_choice == '2':

game = Game(mode=0, starting\_player\_num=random.choice([0, 1]))

game.run()

else:

self.current\_menu = ['easy', 'medium', 'hard', 'back']

self.display()

choice = int(self.user\_choose())

if choice != 4:

game = Game(mode=1, difficulty=self.difficulties[choice - 1],

starting\_player\_num=random.choice([0, 1]))

game.run()

This method asks the user to select a game mode, and if the game mode select is AI vs human, asks them to select the difficulty of the AI. It then runs the game with the settings chosen.

## Menu.load\_game()

def load\_game(self):

last\_save = open('save\_file.txt', 'r')

data = last\_save.readlines()

if len(data) < 13:

print('No save available')

return None

print(data)

save\_state = [x[:-1] for x in data[:-4]]

save\_player\_num = int(data[-4])

save\_prev\_move = (int(data[-3][0]), int(data[-3][1]))

save\_difficulty = None if data[-2] == 'None\n' else int(data[-2])

save\_mode = data[-1]

s

game = Game(mode=save\_mode, difficulty=save\_difficulty, starting\_state=save\_state,

prev\_move=save\_prev\_move, starting\_player\_num=save\_player\_num)

last\_save.close()

game.run()

This method checks if a save is available in the save\_file.txt text file, and if it is, runs the game with all the appropriate settings.

## Menu.run()

The user is asked to selet an option from the main menu, and that option is then run.

def run(self):

while self.quit is False: # when option finished, returns back to # main menu

self.current\_menu = self.main\_menu

self.display()

main\_menu\_option = self.user\_choose()

if main\_menu\_option == '1':

self.new\_game()

elif main\_menu\_option == '2':

self.load\_game()

elif main\_menu\_option == '3':

self.instructions()

elif main\_menu\_option == '4':

quit()

Technical Solution

# Technical Solution- Complex Functions Guide

This is a very vague identification of only a few of the complex functions in my Technical Solution.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Group | Skill | Function Name | Description | Page/Line |
| A | Use of trees |  | A tree structure is one of the key structures in my project, and is stored in the Node.children and Node.parent variables in the Node class, which are accessed by the Monte Carlo class. | Throughout Node and Monte Carlo Class |
| A | Complex user defined use of object orientated programming | Throughout code |  | Throughout code |
| A | Tree traversal | MonteCarlo.select() | Goes through the game tree, selecting the child nodes give the highest UCT values, until a child node is reached. | Line 175 |
| A | Complex user defined algorithms | MonteCarlo class | Contains the entire Monte Carlo Tree Search algorithm | Throughout Monte Carlo class |
| A | Dynamic generation of objects based on complex user-defined use of OOP model. | MonteCarlo.expand()  Menu.new\_game()  Menu.load\_game()  Game.\_\_init\_\_() |  | Lines  197  490  512  309 |
| A | Files organised for direct access | Menu.load\_game()  Game.save\_game()  Menu.tutorial()  Game.clear\_save() |  | Lines  422  436  473  512 |
| B | Multi- dimensional arrays | Throughout project | Throughout project as global grids or game states | Throughout project |
| B | Simple user defined algorithms | Throughout Monte Carlo class |  | Throughout Monte Carlo class |
| C | Simple mathematical calculations | Throughout project |  | Throughout project |

1. **import** math
2. **import** random
4. winners = [(0, 1, 2), (3, 4, 5), (6, 7, 8), (0, 3, 6), (1, 4, 7), (2, 5, 8),
5. (0, 4, 8), (2, 4, 6)]  # different combinations that lead to a local/global win
6. symbols = ['X', 'O']
8. default\_starting\_state = ['         ', '         ', '         ', '         ', '         ',
9. '         ', '         ', '         ', '         ']

12. **def** input\_convertor(coordinate, reverse=False):
13. # Parameters :- coordinate:string, reverse:boolean
14. # Return Type :- tuple
15. # Purpose :- Converts string 2 digit coordinate into keypad tuple or vice versa
16. ###########################
18. conv = [7, 8, 9, 4, 5, 6, 1, 2, 3]
19. x, y = int(coordinate[0]), int(coordinate[1])
21. **if** **not** reverse:
22. acc\_coordinate = (conv.index(x), conv.index(y))
23. **else**:
24. acc\_coordinate = (conv[x], conv[y])
25. **return** acc\_coordinate

28. **def** opposite(dig):
29. # Parameters :- dig:integer (1 or 0)
30. # Return Type :- integer
31. # Purpose :- Returns 1 for 0 and vice versa
32. ###########################
34. **if** dig == 1:
35. **return** 0
36. **else**:
37. **return** 1

40. **def** check\_move(grid, coordinate):
41. # Parameters :- grid: list, coordinate: tuple
42. # Return Type :- boolean
43. # Purpose :- Checks whether or not a move is valid
44. a, b = coordinate
45. **if** len(grid[a]) <= 1:
46. **return** False
48. **if** grid[a][b] != ' ':
49. **return** False
51. **return** True

54. **def** get\_valid\_moves(node\_state, prev\_move):
55. # Parameters :- node\_state:list, prev\_move:tuple
56. # Return Type :- list
57. # Purpose :- Returns a list of all valid child states of a node state
58. valid\_moves = []
59. **if** prev\_move **is** None:
60. **for** a **in** range(9):
61. **for** b **in** range(9):
62. valid\_moves.append((a, b))
64. **else**:
65. x, y = prev\_move
66. **if** ' ' **not** **in** node\_state[y]:
67. **for** a **in** range(9):
68. **for** b **in** range(9):
69. **if** check\_move(node\_state, (a, b)):
70. valid\_moves.append((a, b))
71. **else**:
72. **for** a **in** range(9):
73. **if** check\_move(node\_state, (y, a)):
74. valid\_moves.append((y, a))
76. **return** valid\_moves

79. **def** check\_win(global\_grid):
80. # Parameters :- global\_grid: list
81. # Return Type :- tuple
82. # Purpose :- Checks and returns whether the game state has been won, lost or drawn
83. **for** i **in** range(len(global\_grid)):
84. current\_grid = global\_grid[i]
85. **if** len(current\_grid) > 1:
86. **for** winner **in** winners:
87. a, b, c = winner
88. grid\_winner = current\_grid[a] + current\_grid[b] + current\_grid[c]
89. **if** grid\_winner == 'XXX':
90. global\_grid[i] = 'X'
91. **break**
92. **elif** grid\_winner == 'OOO':
93. global\_grid[i] = 'O'
94. **break**
96. **for** winner **in** winners:
97. a, b, c = winner
98. global\_winner = global\_grid[a] + global\_grid[b] + global\_grid[c]
99. **if** global\_winner == 'OOO':
100. **return** 1, 1
101. **elif** global\_winner == 'XXX':
102. **return** 0, 1
104. **for** grid **in** global\_grid:
105. **if** ' ' **in** grid:
106. **return** None
107. **return** 0.5, 1

110. **class** Node:
111. **def** \_\_init\_\_(self, parent, children, prev\_move, state=None, root=False, value=(0, 0)):
112. self.parent = parent
113. self.children = children
114. self.value = value  # q/Q = wins/ vists
115. self.state = state
116. self.root = root
117. self.UCT = None
118. self.prev\_move = prev\_move
119. self.possible\_moves = get\_valid\_moves(node\_state=self.state, prev\_move=self.prev\_move)
121. **if** root:
122. self.current\_player = 1  # this makes it so that the AI player is O
123. self.depth = 0
124. **else**:
125. self.current\_player = opposite(self.parent.current\_player)
126. self.depth = self.parent.depth + 1
128. **def** display\_node(self):
129. # Parameters :- None
130. # Return Type :- None
131. # Purpose :- Prints the Node.state in a clear format
132. # Explained further in Design
133. state\_copy = self.state[:]
134. **for** grid **in** range(len(state\_copy)):
135. **if** state\_copy[grid] == 'O':
136. state\_copy[grid] = 'OOOOOOOOO'
137. **elif** state\_copy[grid] == 'X':
138. state\_copy[grid] = 'XXXXXXXXX'
140. **print**('╔' + '═' \* 16 + '╗')
141. **for** z **in** range(3):
142. **for** i **in** range(3):
143. **for** grid **in** state\_copy[z \* 3:(z + 1) \* 3]:
144. **print**('║' + grid[i \* 3:(i + 1) \* 3].replace(' ', '#'), end='║')
145. **print**()
146. **if** z == 2:
147. **print**('╚' + '═' \* 16 + '╝')
148. **else**:
149. **print**('╠' + '═' \* 16 + '╣')
150. **print**()

153. **class** MonteCarlo:
154. **def** \_\_init\_\_(self, iterate, grid, prev\_move=None):
155. self.root = Node(parent=None, children=[], state=grid, root=True, prev\_move=prev\_move)
156. self.C = 2 \*\* 0.5
157. self.iterate = iterate
158. self.count = 0
160. **def** get\_UCT(self, node):
161. # Parameters :- node:Node
162. # Return Type :- float
163. # Purpose :- Calculates a UCT value for a Node and returns it
164. **if** node.parent:
165. W = node.value[0]
166. n = node.value[1]
167. N = node.parent.value[1]
168. **if** n == 0:
169. **return** math.inf
170. **else**:
171. **return** W / n + (self.C \* math.sqrt(math.log(N) / n))
172. **else**:
173. **return** math.inf
175. **def** select(self, node):
176. # Parameters :- node:Node
177. # Return Type :- Node
178. # Purpose :- Finds the leaf Node and the pathway to it which maximises the UCT equation
179. **while** len(node.possible\_moves) == 0:
180. **for** child\_node **in** node.children:
181. child\_node.UCT = self.get\_UCT(child\_node)
183. children\_sorted = sorted(node.children, reverse=True, key=**lambda** each\_node: each\_node.UCT)
184. node = children\_sorted[0]
186. # Randomise equal nodes:
187. equal\_UCT\_nodes = []
189. **for** sorted\_node **in** children\_sorted:
190. **if** sorted\_node.UCT == node.UCT:
191. equal\_UCT\_nodes.append(sorted\_node)
193. **if** len(equal\_UCT\_nodes) > 0:
194. node = random.choice(equal\_UCT\_nodes)
195. **return** node
197. **def** expand(self, node):
198. # Parameters :- node:Node
199. # Return Type :- Node
200. # Purpose :- creates and returns a random child node of the node
201. expansion\_move = random.choice(node.possible\_moves)
202. new\_child\_node = self.create\_child\_node(parent=node, move=expansion\_move)
204. **return** new\_child\_node
206. **def** simulation(self, selected\_node):
207. # Parameters :- selected\_node:Node
208. # Return Type :- tuple
209. # Purpose :- Simulates a random game from selected\_node and returns the terminal state: (W, L, D)
210. current\_player = selected\_node.current\_player
211. current\_state = selected\_node.state[:]
212. prev\_move = selected\_node.prev\_move
214. is\_terminal = check\_win(current\_state)
216. **while** is\_terminal **is** None:
217. possible\_moves = get\_valid\_moves(current\_state, prev\_move)
219. random\_move = random.choice(possible\_moves)
221. ra, rb = random\_move
222. current\_state[ra] = current\_state[ra][:rb] + symbols[current\_player] \
223. + current\_state[ra][rb + 1:]
224. is\_terminal = check\_win(current\_state)
225. prev\_move = random\_move
226. current\_player = opposite(current\_player)
228. **return** is\_terminal
230. **def** simulate(self, selected\_node):
231. # Parameters :- selected\_node:Node
232. # Return Type :- None
233. # Purpose :- adds the correct values to the correct Nodes, calls the simulation and back\_propagate methods
234. sim\_result = self.simulation(selected\_node)
236. W1, n1 = selected\_node.value
237. W2, n2 = sim\_result
238. selected\_node.value = (W1 + W2, n1 + n2)
239. self.back\_propagate(selected\_node, (W2, n2))
241. **def** back\_propagate(self, simulated\_node, value):
242. # Parameters :- simulate\_node: Node, value:tuple
243. # Return Type :- None
244. # Purpose :- adds value to the Node.value of each parent of the simulated\_node until root is reached
245. W2, n2 = value
246. **while** simulated\_node.parent **is** **not** None:
247. W1, n1 = simulated\_node.parent.value
248. simulated\_node.parent.value = (W1 + W2, n1 + n2)
249. simulated\_node = simulated\_node.parent
251. **def** Monte\_Carlo(self):
252. # Parameters :- None
253. # Return Type :- Node
254. # Purpose :- Executes the MCTS algorithm and calls the make\_move, returns the best move
255. **while** self.count <= self.iterate:
256. # select the node with the highest UCT value
257. selected\_leaf = self.select(self.root)
259. # if Node hasn't been visited
260. **if** selected\_leaf.value[1] == 0:
261. # simulate a game from the Node and back propagate it
262. self.simulate(selected\_leaf)
264. **else**:  # if Node has been visited
265. # expand the Node
266. simulation\_node = self.expand(selected\_leaf)
268. # and simulate the new Child Node
269. self.simulate(simulation\_node)
271. self.count += 1
273. # AI move is the immediate child node
274. # of the root that has been visited most
275. move\_node = self.make\_move()
277. **return** move\_node
279. **def** make\_move(self):
280. # Parameters :- None
281. # Return Type :- Node
282. # Purpose :- Finds best move Node, returns it
283. max\_visits = 0
284. move\_node = None
286. **for** node **in** self.root.children:
287. **if** node.value[1] >= max\_visits:
288. max\_visits = node.value[1]
289. move\_node = node
291. **return** move\_node
293. **def** create\_child\_node(self, parent, move):
294. # Parameters :- parent:Node, move:tuple
295. # Return Type :- Node
296. # Purpose :- Creates child\_node such that parent+move=child\_node
297. child\_node = Node(parent, [], prev\_move=move, state=parent.state[:])
299. a, b = move
300. child\_node.state[a] = child\_node.state[a][:b] + symbols[parent.current\_player] \
301. + child\_node.state[a][b + 1:]
303. parent.possible\_moves.remove(move)
304. child\_node.parent.children.append(child\_node)
305. **return** child\_node

308. **class** Game:
309. **def** \_\_init\_\_(self, mode, difficulty=None, starting\_state=None, prev\_move=None, starting\_player\_num=random.choice([0, 1])):
310. # starting\_player\_num decides if the human(0) or AI(1) goes first
311. self.mode = int(mode)
312. **if** starting\_state **is** None:
313. starting\_state = ['         ', '         ', '         ', '         ', '         ',
314. '         ', '         ', '         ', '         ']
315. self.starting\_player\_num = starting\_player\_num
316. self.turn = self.starting\_player\_num
317. self.prev\_move = prev\_move
318. self.difficulty = difficulty  # int of iterations of MonteCarlo class AI
319. self.mont = MonteCarlo(iterate=self.difficulty,
320. grid=starting\_state,
321. prev\_move=self.prev\_move)
322. self.result = None
323. self.game\_node = self.mont.root
324. self.players = [['Player 1', 'Player 2'], ['Human', 'AI']]
326. **def** run(self):
327. # Parameters :- None
328. # Return Type :- None
329. # Purpose :- Runs the Ultimate Tic Tac Toe Game in Local Multiplayer or Human vs AI mode
330. # with all selected variables and settings
331. # Full walk through of method available on Design section under Game.run()
332. self.game\_node.display\_node()
333. **if** self.prev\_move **is** **not** None:
334. **print**('previous:', input\_convertor(self.prev\_move, reverse=True))
336. **while** self.result **is** None:
337. **if** self.prev\_move **is** **not** None:
338. self.save\_game()
340. **if** self.turn == 1 **and** self.mode == 1:
341. **print**("AI Turn")
342. **if** self.game\_node.state == default\_starting\_state:
343. self.game\_node = self.mont.create\_child\_node(self.mont.root, (4, 4))
344. self.prev\_move = (4, 4)
345. **else**:
346. self.game\_node = self.mont.Monte\_Carlo()
348. self.prev\_move = self.game\_node.prev\_move
350. **elif** self.turn == 0 **or** self.mode == 0:
351. **print**(self.players[self.mode][self.turn] + ' Turn')
352. human\_move = input('Coordinates: ')
353. **if** human\_move == '000':
354. **print**('Game state saved and game aborted')
355. **return** None
357. **while** self.handle\_input(human\_move) **is** False:
358. **print**("Invalid Move!")
359. human\_move = input('Coordinates: ')
360. **if** human\_move == '000':
361. **print**('Game state saved and game aborted')
362. **return** None
364. a, b = input\_convertor(human\_move, reverse=False)
366. self.game\_node.state[a] = self.game\_node.state[a][:b] + symbols[self.turn] + self.game\_node.state[a][b + 1:]
367. self.prev\_move = (a, b)
368. self.game\_node.prev\_move = (a, b)
370. self.result = check\_win(self.game\_node.state)
372. self.game\_node.display\_node()
374. **print**('previous:', input\_convertor(self.prev\_move, reverse=True))
376. **if** self.mode == 1:
377. self.mont.\_\_init\_\_(grid=self.game\_node.state, iterate=self.difficulty, prev\_move=self.prev\_move)
379. self.turn = opposite(self.turn)
380. self.endgame()
381. self.clear\_save()
383. **def** handle\_input(self, coordinate):
384. # Parameters :- coordinate:tuple
385. # Return Type :- Boolean
386. # Purpose :- check if the input coordinate is valid or not
387. **if** coordinate == '000':
388. **return** None
390. **if** len(coordinate) != 2:
391. **return** False
393. **for** val **in** coordinate:
394. **if** val **not** **in** ['1', '2', '3', '4', '5', '6', '7', '8', '9']:
395. **return** False
397. **if** int(coordinate) > 100 **or** int(coordinate) < 1:
398. **return** False
400. a, b = input\_convertor(coordinate)
402. move\_check = check\_move(self.mont.root.state, (a, b))
403. **if** move\_check **is** False:
404. **return** move\_check
406. **if** self.prev\_move **is** **not** None:
407. **if** len(self.game\_node.state[self.game\_node.prev\_move[1]]) != 1 \
408. **and** a != self.game\_node.prev\_move[1]:
409. **return** False
411. **def** endgame(self):
412. # Parameters :- None
413. # Return Type :- None
414. # Purpose :- Displays a Win/Loss/Draw string at end of game
415. **if** self.result == (1, 1):
416. **print**("{} Won!".format(self.players[self.mode][opposite(self.turn)]))
417. **elif** self.result == (0, 1):
418. **print**("{} Won!".format(self.players[self.mode][opposite(self.turn)]))
419. **else**:
420. **print**("Draw!")
422. **def** save\_game(self):
423. # Parameters :- None
424. # Return Type :- None
425. # Purpose :- Saves the game variables and state in text file:
426. # 'save\_file.txt' to be loaded from later
427. save\_file = open('save\_file.txt', 'w')
428. **for** local\_grid **in** self.game\_node.state:
429. save\_file.write(local\_grid + '\n')
430. save\_file.write(str(self.turn) + '\n')
431. save\_file.write(str(self.prev\_move[0]) + str(self.prev\_move[1]) + '\n')
432. save\_file.write(str(self.difficulty) + '\n')
433. save\_file.write(str(self.mode))
434. save\_file.close()
436. **def** clear\_save(self):
437. # Parameters :- None
438. # Return Type :- None
439. # Purpose :- Erases the contents of text file 'save\_file.txt'
440. # if game has ended
441. open('save\_file.txt', 'w').close()

444. **class** Menu:
445. **def** \_\_init\_\_(self):
446. self.main\_menu = ['New Game', 'Load Game', 'Instructions', 'Exit']
447. self.current\_menu = self.main\_menu
448. self.quit = False
449. self.difficulties = [100, 1000, 5000]
451. **def** display(self):
452. # Parameters :- None
453. # Return Type :- None
454. # Purpose :- Displays all self.current menu options in the format:
455. # 1 - option 1
456. # 2 - option 2 etc.
457. **for** i **in** range(len(self.current\_menu)):
458. **print**(i + 1, '-', self.current\_menu[i])
460. **def** user\_choose(self):
461. # Parameters :- None
462. # Return Type :- string
463. # Purpose :- Gets a valid user option choice
464. # from the self.current menu options
465. option = input(': ')
467. **while** option **not** **in** [str(x) **for** x **in** range(1, len(self.current\_menu) + 1)]:
468. **print**('Invalid')
469. option = input(': ')
471. **return** option
473. **def** instructions(self):
474. # Parameters :- None
475. # Return Type :- None
476. # Purpose :- Displays the instructions from instructions.txt
477. # in an interactive way, so that after every instructions page, waits for input
478. # allowing user time to read the instructions page
479. instructions\_file = open('instructions.txt', 'r', encoding='utf8')
480. instruction\_lines = instructions\_file.readlines()
481. **for** line **in** instruction\_lines:
482. **if** 'Page' **in** line:
483. **if** input(':') == '0':
484. **print**()
485. **break**
486. **print**(line, end='')
487. **print**()
488. instructions\_file.close()
490. **def** new\_game(self):
491. # Parameters :- None
492. # Return Type :- None
493. # Purpose :- Gets a valid user option of game time,
494. # and then if applicable, of AI difficulty
495. # then runs an Ultimate Tic Tac Toe Game
496. # with the user chosen settings and variables
497. self.current\_menu = ['Human vs AI', 'Local Multiplayer']
498. self.display()
499. mode\_choice = self.user\_choose()
500. **if** mode\_choice == '2':
501. game = Game(mode=0, starting\_player\_num=random.choice([0, 1]))
502. game.run()
503. **else**:
504. self.current\_menu = ['easy', 'medium', 'hard', 'back']
505. self.display()
506. choice = int(self.user\_choose())
507. **if** choice != 4:
508. game = Game(mode=1, difficulty=self.difficulties[choice - 1],
509. starting\_player\_num=random.choice([0, 1]))
510. game.run()
512. **def** load\_game(self):
513. # Parameters :- None
514. # Return Type :- None
515. # Purpose :- Opens the save\_file.txt and uses the stored data
516. # to start a game of ultimate tic tac toe
517. # so that there is no difference between the loaded game
518. # and the original saved game
519. last\_save = open('save\_file.txt', 'r')
520. data = last\_save.readlines()
521. **if** len(data) < 13:
522. **print**('No save available')
523. **return** None
524. **print**(data)
525. save\_state = [x[:-1] **for** x **in** data[:-4]]
526. save\_player\_num = int(data[-4])
527. save\_prev\_move = (int(data[-3][0]), int(data[-3][1]))
528. save\_difficulty = None **if** data[-2] == 'None\n' **else** int(data[-2])
529. save\_mode = data[-1]
531. game = Game(mode=save\_mode, difficulty=save\_difficulty, starting\_state=save\_state,
532. prev\_move=save\_prev\_move, starting\_player\_num=save\_player\_num)
533. last\_save.close()
534. game.run()
536. **def** run(self):
537. # Parameters :- None
538. # Return Type :- None
539. # Purpose :- Runs all menus,
540. # and if 4 is chosen in main menu, quit program
541. **while** self.quit **is** False:
542. self.current\_menu = self.main\_menu
543. self.display()
544. main\_menu\_option = self.user\_choose()
546. **if** main\_menu\_option == '1':
547. self.new\_game()
548. **elif** main\_menu\_option == '2':
549. self.load\_game()
550. **elif** main\_menu\_option == '3':
551. self.instructions()
552. **elif** main\_menu\_option == '4':
553. quit()

556. menu = Menu()
557. menu.run()

Testing

Testing is required for my project as it ensures everything works as expected.

My Local Multiplayer Ultimate Tic Tac Toe AI project has 5 main parts which require testing: Menus, Instructions, Loading/Saving Games, Gameplay in AI mode, Gameplay in Local Multiplayer Mode, and the AI itself.

## 1: Menus

To test the main menu, I will be trying valid and invalid inputs and checking if they produce the expected outcome, and I will check if the game exit function works.

To test the difficulty menu, which appears after selecting the ‘new game’ option, I will be trying valid and invalid inputs, and checking that they produce the expected outcome, and I will also check whether the ‘back’ option returns the user back to the main menu.

This section will be testing my Menu class and all its functions, and seeing if everything in them works as it should.

## 2: Instructions

In order to test the ‘Instructions’ page, I will see if the ‘back’ input option returns me back to the main menu, and if the enter key takes me to the next page of instructions or not. I will be checking different inputs to see if they produce the expected results.

This section will also be testing the Menu class, particularly the instructions() method of the class.

## 3: Loading/Saving

The game is expected to be saved after every turn, until the game reaches a terminal state (win, loss or draw). I will check that the game can successfully be saved and loaded from the last save, with the correct settings and global grid, and that the load and save can occur during the turn of both the AI or the human player. I will check that an old save can be successfully overwritten by a new save.

This will be testing the Game.load(), Game.clear\_save(), and Menu.load\_game() methods.

## 4: Gameplay – Local Multiplayer Mode

I will check that the global grid and previous move are displayed correctly and that the turn progression is correct for local multiplayer mode (Player 1 and Player 2).I will check that there is a 50% chance for either the first player or second player to go first in the game. I will ensure that the quit game option works correctly and that winning/losing/drawing ends the game, and causes the correct message to be displayed. I will also check to see that each local grid is checked correctly for win/loss.

This will test the Game class, and all its methods and if it handles game modes correctly.

This will also be testing the check\_win(), check\_move(), and all other global functions.

## 5: Gameplay – AI Mode

I will check that the global grid is displayed correctly, that the previous move is displayed correctly, and that the turn progression is correct. I will also test the input handling by inputting invalid and valid moves. I will check that the quit game option works correctly. I will also check to see that each local grid is checked correctly for win/loss and that winning/losing/drawing ends the game in AI mode and causes the correct message to be displayed.

This will be testing the Game class and MonteCarlo Class as well as the Node Class, and all the methods within these classes. This will also test all the global functions as they are related to gameplay.

## 6: AI

I will check that the more difficult AI is genuinely more difficult to play against, by playing the Easy AI against the Medium and the Medium AI against the Hard. This will prove that there is a true progression in difficulty between the 3 AI.

I will also play the Hard AI against an online hard AI to show that the AI works as well as other well-known AI products.

This section will be testing the Monte Carlo Class, Node Class, Game Class, and all global functions and check that the AI works at a product standard and that there is a clear difference between the different difficulties of AI.

## Types of Invalid Moves:

Invalid moves here refers to inputs in the Gameplay aspects of the project which are correct inputs but are rendered invalid by the rules of the game.

These fall into 3 categories:

Type 1 – Where the player attempts to place their symbol at a location that already contains a symbol.

Type 2 – Where the player attempts to place their symbol at a local grid which is already full i.e. has been won/lost/drawn.

Type 3 – Where the player attempts to place their symbol in a local grid which does not correspond to the previous player’s move (and the previous player’s move does not correspond to a full local grid).

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test | Purpose  The first number in ‘Test’ corresponds to the section (1-6 above) being testing for. | Test Data/Input | Expected Outcome | Outcome | Changes Required | Timestamp of Test |
| 1.1 | Testing that invalid main menu inputs are handled. | 100, 10, -1, =, 0, fgfdgdf, ENTER key | Output ‘Invalid’ and wait for valid input. | Outputted ‘Invalid’ and waited for valid input. | None | 0:00 – 0:30 |
| 1.2 | Testing that the exit main menu option stops the application. | 4 | Stop the application from running, and exit. | Stopped the application from running, and exit. | None | 0:30 – 0:32 |
| 1.3 | Testing that invalid new game menu inputs are handled. | fdg, 4353, 3, 0, -1, -12443, ENTER key, @{:}@ | Output ‘Invalid’ and wait for valid input. | Outputted ‘Invalid’ and waited for valid input. | None | 0:32 – 1:00 |
| 1.4 | Testing that invalid AI difficulty menu inputs are handled. | 0, -1, 5, 32432, fssdfsdg, @:{, ENTER key | Output ‘Invalid’ and wait for valid input. | Outputted ‘Invalid’ and waited for valid input. | None | 1:00 – 1:24 |
| 1.5 | Testing that all main menu options produce the correct result. | 1, 2, 3, 4 | 1 takes the user to the ‘new game options’ page.  2 loads a saved game or displays an error if there is no saved game.  3 takes the user to the instructions page.  4 exits the game. | 1 takes the user to the ‘new game options’ page.  2 loads a saved game or displays an error if there is no saved game.  3 takes the user to the instructions page.  4 exits the game. | None | 1:24 – 1:40 |
| 1.6 | Testing that all new game menu options produce the correct result. | 1, 2 | 1 takes the user to the AI difficulty menu, and 2 starts a local multiplayer game. | 1 takes the user to the AI difficulty menu, and 2 starts a local multiplayer game. | None | 1:40 – 2:21 |
| 1.7 | Testing that all AI difficulty menu options produce the correct result. | 1, 2, 3, 4 | 1 starts a human vs easy AI game,  2 starts a human vs Medium AI game,  3 starts a human vs Hard AI game,  4 takes the user back to the main menu. | 1 starts a human vs easy AI game,  2 starts a human vs Medium AI game,  3 starts a human vs Hard AI game,  4 takes the user back to the main menu. | None | 1:40 – 2:08 |
| 2.1 | Testing that inputting ‘0’ takes user to the main menu page. | 0 | Inputting 0 should take the user back to the main menu page. | Inputting 0 takes the user back to the main menu page. | None | 2:21 – 2:25 |
| 2.2 | Testing that the ENTER key causes the correct navigation. | ENTER key | Pressing enter takes the user to the next page of the instructions, until instruction pages are complete, where it then takes the user back to the main menu page. | Pressing enter takes the user to the next page of the instructions, until instruction pages are complete, where it then takes the user back to the main menu page. | None | 2:25 – 2:50 |
| 2.3 | Testing that all inputs cause the correct navigation from the instructions page. | HGF, 56756DFS\*?// | Inputting any characters should take the user to the next instructions page until instructions pages are complete, where it then takes the user back to the main menu page. | Inputting any characters takes the user to the next instructions page until instructions pages are complete, where it then takes the user back to the main menu page. | None | 2:50 – 3:08 |
| 3.1 | Test that if no save is available, displays error message and handles accordingly. | Input ‘2’ in main menu. | Display message ‘No save available’ and return the user back to the main menu. | Displays message ‘No save available’ and returns the user back to the main menu. | None | 3:08 – 3:17 |
| 3.2 | Check that game saves and loads correctly in human vs AI mode. | Create game of human vs AI in different difficulty modes, where the next turn is the AI and the Human’s turn, and where the global grid is randomly mixed, and then load this global grid to see if it loads correctly.  (specifics can be seen in Test Video) | Load the game correctly so that the following are the same in the loaded game as in the original:  - current player’s turn  - global grid  -previous move  - AI difficulty | Loads the game correctly so that the following are the same in the loaded game as in the original:  - current player’s turn  - global grid  -previous move  - AI difficulty | None | 3:17 – 5:20 |
| 3.3 | Check that game saves and loads correctly in Local Multiplayer mode. | Create game of Local Multiplayer AI, where the next turn is Player 1 and Player 2’s turn, and where the global grid is randomly mixed, and then load this global grid to see if it loads correctly.  (specifics can be seen in Test Video) | Saves the game by saving the state of the game to a text file.  Load the game from the text file correctly so that the following are the same in the loaded game as in the original:  - current player’s turn  - global grid  -previous move | Loads the game correctly so that the following are the same in the loaded game as in the original:  - current player’s turn  - global grid  -previous move  - AI difficulty | None | 5:20 – 6:14 |
| 3.4 | Check that old save is overwritten by new saves. | Create a new game in any mode, randomise the global grid by playing the game randomly for a short period, then save and exit from the game.  Load the game to see if it has saved, then repeat with a different global grid, current turn and previous move, and on main menu select ‘load game’ to see if the older game save has been overwritten. | The 1st game save is successfully overwritten by the newest game save, and the newest game save is therefore run. | The 1st game save was successfully overwritten by the newest game save, and the newest game save ran. | None | 6:14 – 7:13 |
| 4.1 | Test that the quit input quits the game. | ‘000’ | A message is displayed saying ‘Global grid saved and game aborted’, the game is saved, and the user is taken back to the main menu. | A message is displayed saying ‘Global grid saved and game aborted’, the game is saved, and the user is taken back to the main menu. | None | 7:13 – 7:21 |
| 4.2 | Check that there is a 50% chance for either player to start the local multiplayer game. | Run a local multiplayer game multiple times to show that both player 1 and player 2 can have the first move. | After running the local multiplayer game, approximately half the time, player 1 has the first move, and half the time, player 2 has the first move. | After running the local multiplayer game, approximately half the time, player 1 has the first move, and half the time, player 2 has the first move. | None | 7:21 – 7:52 |
| 4.3 | Test that incorrect inputs are handled correctly. | DSF, sdf, 3445435, 0, -1, fsd45324, -213123, 1010, @{}dfs234, ENTER key | The message ‘Invalid Move!’ is displayed and the user is asked for Coordinates again, this repeats until a valid move or command is given. | The message ‘Invalid Move!’ is displayed and the user is asked for Coordinates again, this repeats until a valid move or command is given. | None | 7:52 – 8:30 |
| 4.4 | Check that the current player is displayed correctly. |  | The current player is displayed correctly in game in the format ‘Player x Turn’ where x cycles through values 1 and 2 after every turn. | The current player is displayed correctly in game in the format ‘Player x Turn’ where x cycles through values 1 and 2 after every turn. | None | 7:13 – 7:21 |
| 4.5 | Check that the global grid is displayed correctly. |  | The global grid is displayed in the format shown in the Design section, with ‘O’ or ‘X’ replacing the ‘#’ where users made their move. | The global grid is displayed in the format shown in the Design section, with ‘O’ or ‘X’ replacing the ‘#’ where users made their move. | None | 7:13 – 7:21 |
| 4.6.1 | Check that invalid move Type 1 is handled. | ‘55’ | Shows message ‘Invalid Move!’ and the user is asked for a valid Coordinate or command with the message ‘Coordinates: ’ until a valid coordinate or command is entered. | Shows message ‘Invalid Move!’ and the user is asked for a valid Coordinate or command with the message ‘Coordinates: ’ until a valid coordinate or command is entered. | None | 8:30 – 8:38 |
| 4.6.2 | Check that invalid move Type 2 is handled. | 52, 53, 57 | Shows message ‘Invalid Move!’ and the user is asked for a valid Coordinate or command with the message ‘Coordinates: ’ until a valid coordinate or command is entered. | Shows message ‘Invalid Move!’ and the user is asked for a valid Coordinate or command with the message ‘Coordinates: ’ until a valid coordinate or command is entered. |  | 9:18 – 9:49 |
| 4.6.3 | Check that invalid move Type 3 is handled. | 25  15  65 | Shows message ‘Invalid Move!’ and the user is asked for a valid Coordinate or command with the message ‘Coordinates: ’ until a valid coordinate or command is entered. | Shows message ‘Invalid Move!’ and the user is asked for a valid Coordinate or command with the message ‘Coordinates: ’ until a valid coordinate or command is entered. | None | 8:38 – 8:46 |
| 4.7 | Check that player can move their symbol anywhere if the previous move refers to a full local grid. | 99 | Any coordinate with values between 11-99 inclusive is accepted, without an invalid coordinate message, and is shown on the global grid and on the previous move message. | Any coordinate with values between 11-99 inclusive is accepted, without an invalid coordinate message, and is shown on the global grid and on the previous move message. | None | 9:05 – 9:18 |
| 4.8.1 | Check that winning/losing a local grid produces the desired result. | 51 | A won local board should be marked by the local grid’s values changing to the player’s symbol. | The won local board is marked by the local grid’s values changing to the player’s symbol. (X in first timestamp, O in second) |  | 8:46 – 9:05 &  9:48 – 10:27 |
| 4.8.2 | Check that a local grid resulting in a draw produces the desired result. |  | A drawn local grid is shown as full of symbols in the appropriate locations chosen by the players. | A drawn local grid is shown as full of symbols in the appropriate locations chosen by the players. | None | 10:57 – 11:55 |
| 4.9.1 | Check that winning the global grid produces the desired result. | 94 | Winning the global grid produces the message ‘Player x Won!’ where x is the winning player, and takes the user back to the main menu. | Winning the global grid produces the message ‘Player x Won!’ where x is the winning player, and takes the user back to the main menu. | None | 11:55 – 12:46 |
| 4.9.2 | Check that losing the global grid produces the desired result. | 69 | Losing the global grid produces the message ‘Player x Won!’ where x is the winning player, and takes the user back to the main menu. | Losing the global grid produces the message ‘Player x Won!’ where x is the winning player, and takes the user back to the main menu. |  | 12:46 – 13:27 |
| 4.9.3 | Check that drawing the global grid produces the desired result. | 31 | When the global grid is drawn, the message ‘Draw!’ is produced and the user is taken back to the main menu. | When the global grid is drawn, the message ‘Draw!’ is produced and the user is taken back to the main menu. | None | 13:27 – 13:46 |
| 5.1 | Test that the quit input quits the game. | 000 | Inputting 000 aborts the current game and saves it, and returns the user back to the main menu. | Inputting 000 aborts the current game and saves it, and returns the user back to the main menu. | None | 13:46 – 13:53 |
| 5.2 | Check that there is a 50% chance for either player to start the human vs AI game. | Run a human vs AI game multiple times to show that both player 1 and player 2 can have the first move. | After running the human vs AI game, approximately half the time, Human Player has the first move, and half the time, AI has the first move. | After running the human vs AI game, approximately half the time, Human Player has the first move, and half the time, AI has the first move. | None | 13:53 – 14:23 |
| 5.3 | Test that incorrect inputs are handled. | 0, jkljkkjl, 0, 00, -1, -213, @:}, dsfds324, 34sdf, ENTER key | The message ‘Invalid Move!’ is displayed and the user is asked for Coordinates again, this repeats until a valid move or command is given. | The message ‘Invalid Move!’ is displayed and the user is asked for Coordinates again, this repeats until a valid move or command is given. | None | 14:23 – 15:02 |
| 5.4 | Check that the global grid is displayed correctly. |  | The global grid is displayed in the format shown in the Design section, with ‘O’ or ‘X’ replacing the ‘#’ where users made their move. | The global grid is displayed in the format shown in the Design section, with ‘O’ or ‘X’ replacing the ‘#’ where users made their move. | None | 13:46 – 13:53 |
| 5.5 | Check that the player is displayed correctly. |  | The current player is displayed correctly in game in the format ‘Player x Turn’ where x cycles through values 1 and 2 after every turn. | The current player is displayed correctly in game in the format ‘Player x Turn’ where x cycles through values 1 and 2 after every turn. | None | 13:46 – 13:53 |
| 5.6.1 | Check that invalid move Type 1 is handled. |  | Shows message ‘Invalid Move!’ and the user is asked for a valid Coordinate or command with the message ‘Coordinates: ’ until a valid coordinate or command is entered. | Shows message ‘Invalid Move!’ and the user is asked for a valid Coordinate or command with the message ‘Coordinates: ’ until a valid coordinate or command is entered. | None | 15:31 – 15:45 |
| 5.6.2 | Check that invalid move Type 2 is handled. |  | Shows message ‘Invalid Move!’ and the user is asked for a valid Coordinate or command with the message ‘Coordinates: ’ until a valid coordinate or command is entered. | Shows message ‘Invalid Move!’ and the user is asked for a valid Coordinate or command with the message ‘Coordinates: ’ until a valid coordinate or command is entered. |  | 18:10 – 18:25 |
| 5.6.3 | Check that invalid move Type 3 is handled. |  | Shows message ‘Invalid Move!’ and the user is asked for a valid Coordinate or command with the message ‘Coordinates: ’ until a valid coordinate or command is entered. | Shows message ‘Invalid Move!’ and the user is asked for a valid Coordinate or command with the message ‘Coordinates: ’ until a valid coordinate or command is entered. | None | 15:02 – 15:31 |
| 5.7 | Check that player can move their symbol anywhere if the previous move refers to a full local grid. |  | Any coordinate with values between 11-99 inclusive is accepted, without an invalid coordinate message, and is shown on the global grid and on the previous move message. | Any coordinate with values between 11-99 inclusive is accepted, without an invalid coordinate message, and is shown on the global grid and on the previous move message. | None | 16:23 – 16:49 |
| 5.8 | Check that winning/losing/drawing a local grid produces the desired result. |  | A won/loss local grid should be marked by the local grid’s values changing to the player’s/opponent’s symbol. | A won/loss local grid is marked by the local grid’s values changing to the player’s/opponent’s symbol. | None | 15:45 – 16:23 |
| 5.9 | Check that winning/losing/drawing the global grid produces the desired result. |  | Winning the global grid produces the message ‘Player x Won!’ where x is the winning player, and takes the user back to the main menu. | Winning the global grid produces the message ‘Player x Won!’ where x is the winning player, and takes the user back to the main menu. | None | 18:25 – 18:59 |
| 6.1 | Play the Medium AI against the Easy AI to check that the Medium is better. |  | The Medium AI should beat the Easy AI. | The Medium AI beats the easy AI. | None | 18:59 – 21:24 |
| 6.2 | Play the Hard AI against the Medium AI to check the Hard is better. |  | The Hard AI should beat the Medium AI. | The Hard AI beats the Medium AI. | None | 21:24 – 26:05 |
| 6.3 | Play the Hard AI against an online Hard AI to check that this project’s Hard AI is as good as other well-known products. |  | The Hard AI should draw with or beat the online Hard AI. | The Hard AI beats the online Hard AI. | None | 26:05 – 31:22 |

# Testing Video

<https://youtu.be/m04pUdzfURg>

Evaluation

# Objective Analysis

|  |  |  |  |
| --- | --- | --- | --- |
| Objective Reference  (See Analysis Section Objectives) | Met? | Comments | Testing Reference  (See Testing Section) |
| 1.1 – 1.8 | All met except 1.6 | 1.6 could not be met as there was not enough time to create a full GUI which extends to every part of the game. | 1.1, 1.2 and 1.5 |
| 2.1 – 2.6 | All met |  | 2.1 – 2.3 |
| 3.1 – 3.5 | All met |  | 1.3 and 1.6 |
| 4.1 – 4.4 | All met |  | 1.4 and 1.7 |
| 5.1 – 5.4 | All met | Only Objective 5.4 can be tested directly, Test references for which can be found to the right.  Objective 5.1 – 5.3 can be tested indirectly throughout the Testing section. | Objective 5.4:  4.2 and 5.2  Objectives 5.1-5.3:  Tests 4.1 – 4.9.3 and  Tests 5.1 – 5.9 and  Tests 6.1 – 6.3 |
| 6.1 – 6.3 | All met |  | 6.1 – 6.3 and  1.7 |
| 7.1 – 7.2 | All met |  | 5.5 and 5.9 |
| 8.1 – 8.2 | All met |  | 4.9.1 – 4.9.3 and  4.4 |
| 9.1 – 9.4 | All met | 9.1 cannot be tested directly but can be tested indirectly and is tested indirectly throughout the testing video. | 4.4-4.5 and  5.4-5.5 |
| 10.1 – 10.5 | All met | 10.5 is tested indirectly throughout the testing video. | 5.1 and 5.3 and  5.7 and 4.1 and 4.3 and 4.7 |
| 11.1 – 11.4 | All met |  | 5.6.1 – 5.6.3 and  4.6.1 – 4.6.3 |
| 12.1 – 12.4 | All met | A global grid draw cannot be realistically tested for in AI mode since it would be extremely improbable and unrealistic, especially since AI moves are not controlled by the player. However, drawing is tested in the local multiplayer game mode, and since both modes use the same game system, global grid drawing must work in both cases. | 5.8 - 5.9 and  4.8.1 – 4.8.2 and  4.9.1 – 4.9.3 |
| 13.1 – 13.2 | All met |  | 4.9.1 – 4.9.3 and  5.9 |
| 14.1 – 14.7 | All met |  | 3.1 – 3.4 |
| 15 - 22 | All met | None of these objectives can be tested for directly but all are tested for indirectly throughout the Human vs AI mode parts of the testing video, since the AI would not work had these objectives not been implemented. | 6.1 – 6.3 and all human vs AI mode gameplay parts of testing video. |

# End-User Feedback:

## How easy is the application to use?

“I found it very easy to use. I have never played a game of Ultimate Tic Tac Toe, and after reading the well written instructions, I understood how to play the game.  
Every option in the game was numbered, so I found it very easy to navigate.

The move coordinate inputting system which uses the key-pad was very intuitive, and made it so I did not have to think at all about how to input the move, and only about which move to input.

The game saves after every turn, which is extremely helpful since it means that if I close it by accident and lose my progress, I can simply load the saved game.

The game displays the previous move after every turn, which makes it very easy to figure out what the last move was in case you lost track.

My only annoyance is that during the gameplay the way to exit is by inputting ‘000’ but this is only mentioned in the instructions, not at the start of games, so if a player forgets, it can become frustrating.”

## Does the application fulfil the objectives, as shown in the table?

“After having read the table, and the objectives it refers to, I agree with the table that all objectives are met apart from the creation of a GUI. The game is also quite challenging and especially fun when played with AI at hard difficulty.”

## Any criticisms?

“My only criticisms are the lack of a GUI, which would be far more pleasing to look at, and the lack of a clear message at the start of every game explaining that ‘000’ must be used to exit the game.

Apart from these, I found the application to be very satisfactory for its purpose. I found no errors or bugs while running the application.”

## Any improvements or extensions?

“I would suggest creating a GUI, which will be a lot more pleasing to look at and interact with. I would also suggest putting an explanation at the start of every game that ‘000’ must be used to exit the game. Apart from these, I would like to be able to play with other players around the world in an online multiplayer option. I would also like to be able to run the application without having to download python.”

# Analysis of end-user feedback

The feedback has shown that despite being in a text based UI format, this application is extremely easy to use, and that the input methods are intuitive, easy, and well thought out. I am happy to see that the user found the gameplay fun and challenging, and that everything worked without any bugs or errors. This feedback has given me confidence that this application has fulfilled all the objectives apart from objective 1.6, which is to build a GUI.

The end user has also pointed out a small flaw in not including a clear explanation at the start of every game on how to exit the game, and has pointed out that a GUI would be much more pleasing to look at and interact with. The end user also said that they would like to see an online multiplayer option to the game, and the ability to run the game without python. These ideas and extensions are discussed below.

# Possible extensions

If I had the ability to start my program all over again, I would include the following changes:

Firstly, I would create a GUI, as my end user suggested, instead of a text-based interface, with colours, music etc. As this would increase the appeal of my program and make it more enjoyable to interact with. I would do this using a python library called PyQt which can be used to create GUIs fairly easily. I would then colour the different player symbols with contrasting colours and highlight the last move symbol so it is very easy to see. I would also highlight the local grids the current player is allowed to play in so there is no confusion.

In order to make my program executable without having to download python, I could use a python library to convert the python file into a windows .exe file, which can then be run.

Secondly, I would include more difficulty modes for my AI, which I had planned to do from the start, but I had found that if I increased the number of iterations too much to make the AI more difficult, the AI would take too long to make a turn. This means if I want to include more difficulties, I would have to rework core aspects of my AI code to make it more efficient and produce results in a smaller amount of time.

Adding more difficult AI modes makes the game more challenging and fun.

As my end user also suggested, I would like to have an online multiplayer game option, which would require the use of networking, and programming a server client system. This would make my game more fun since it would mean you would not be dependant on playing local multiplayer with humans in the same area as you- you would be able to play with players around the world as well as against AI.

An extension to this idea that I have thought of is incorporating a player rank, which would be accessible from the application in the main menu. I would then match up the online multiplayer players against each other based on their ranks, and reward them if they win by giving them higher ranks on the player rank page. This would make the game more competitive, challenging and therefore more fun.

Lastly, as my end user suggested for the text based version, I would explicitly state that ‘000’ is used to exit the game at the start of every game to ensure there is no confusion.