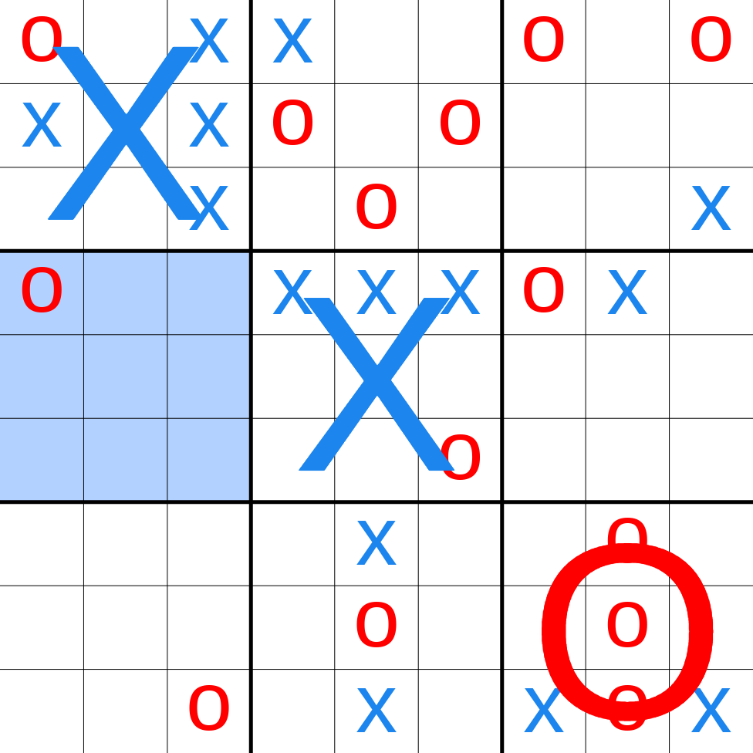
Analysis

# 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.

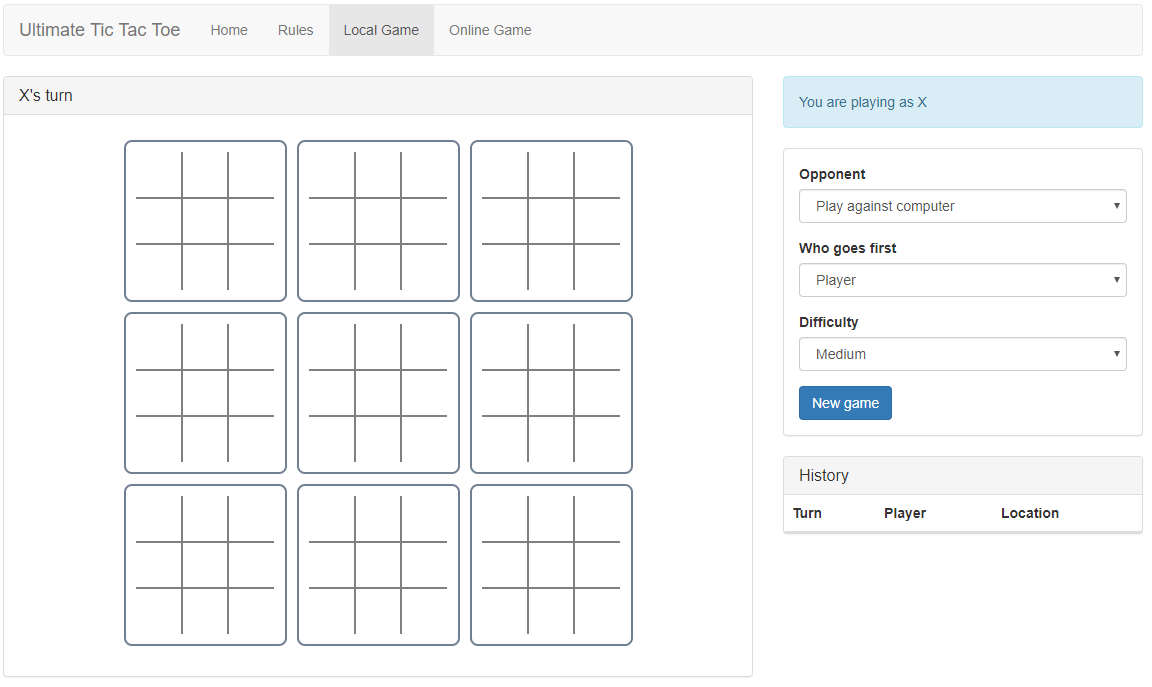
Ultimate Tic Tac Toe is a difficult game to grasp and play strategically, hence it is difficult to find players near your skill level. Furthermore, Ultimate Tic Tac Toe is not very famous because of its higher complexity and the inability to play it recreationally on paper as with Tic Tac Toe.

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 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.

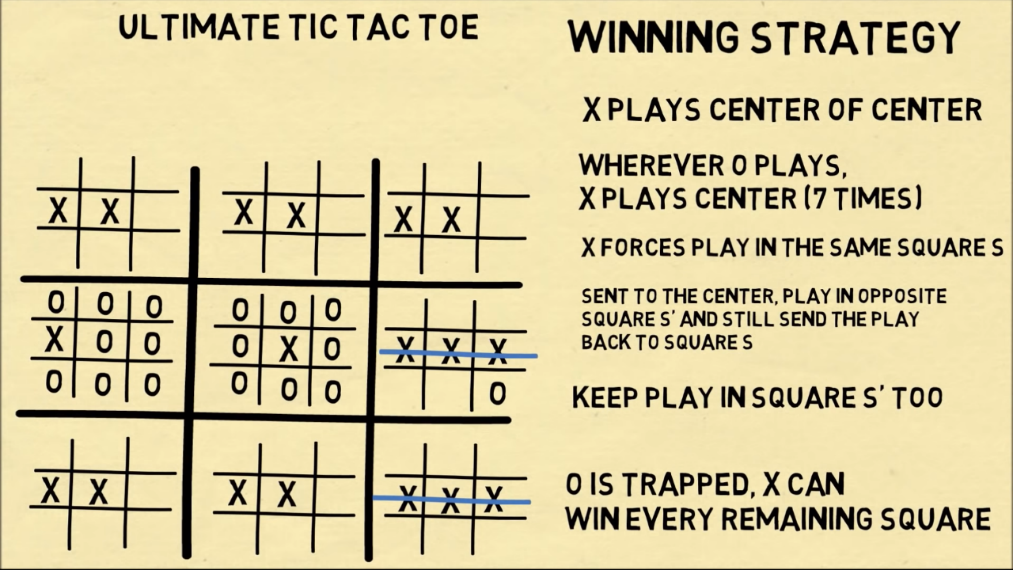
### Rule Variation:

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

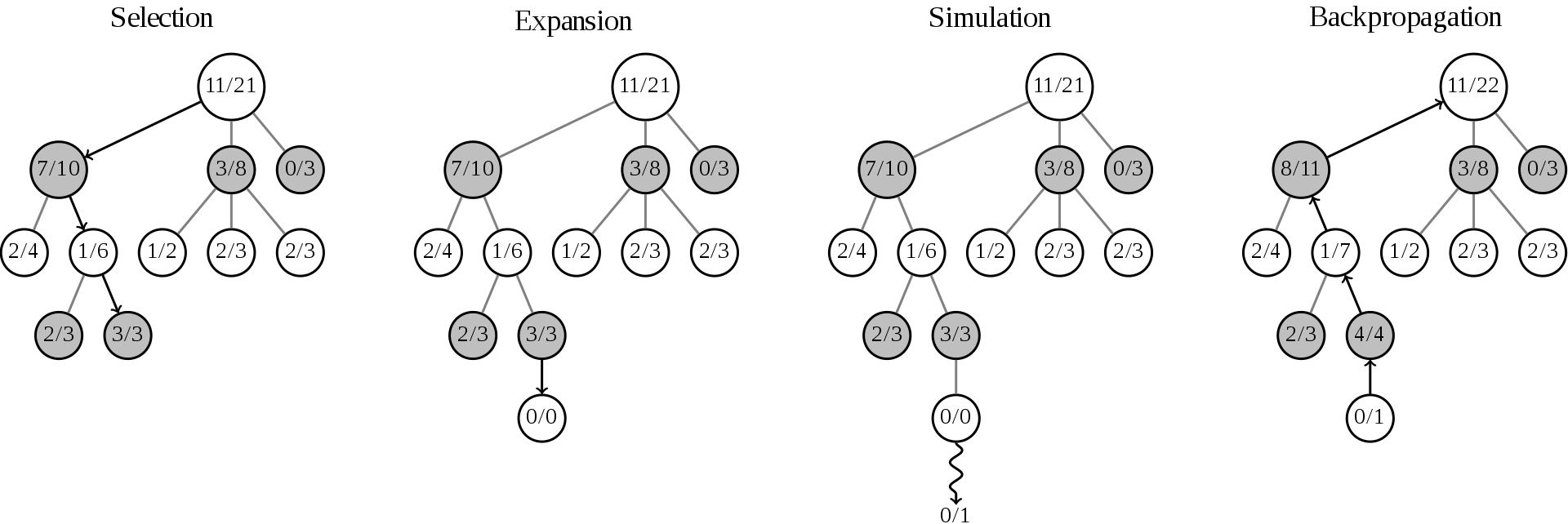
Ultimate Tic Tac Toe is significantly more complex than other variations of Tic Tac Toe.

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.)

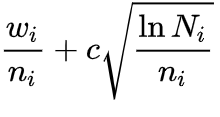
This process continues until either a set amount of time has been reached, or until a given number of iterations have been reached.

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