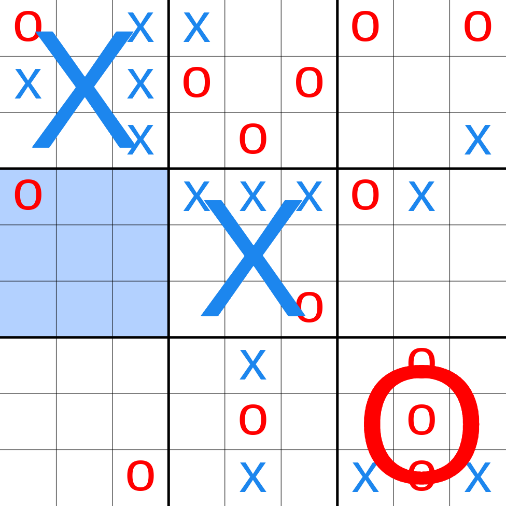
Analysis

# 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 online multiplayer and co-op (multiplayer on same device) modes.

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.

Global Grid: larger grid on outside, or Green



Local Grid: smaller grids, or Dark Blue

Player Symbols: X or O

Ultimate Tic Tac Toe consists of a 3 by 3 global TTT 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 corresponds to the local grid the next player is able to position their symbol in.

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

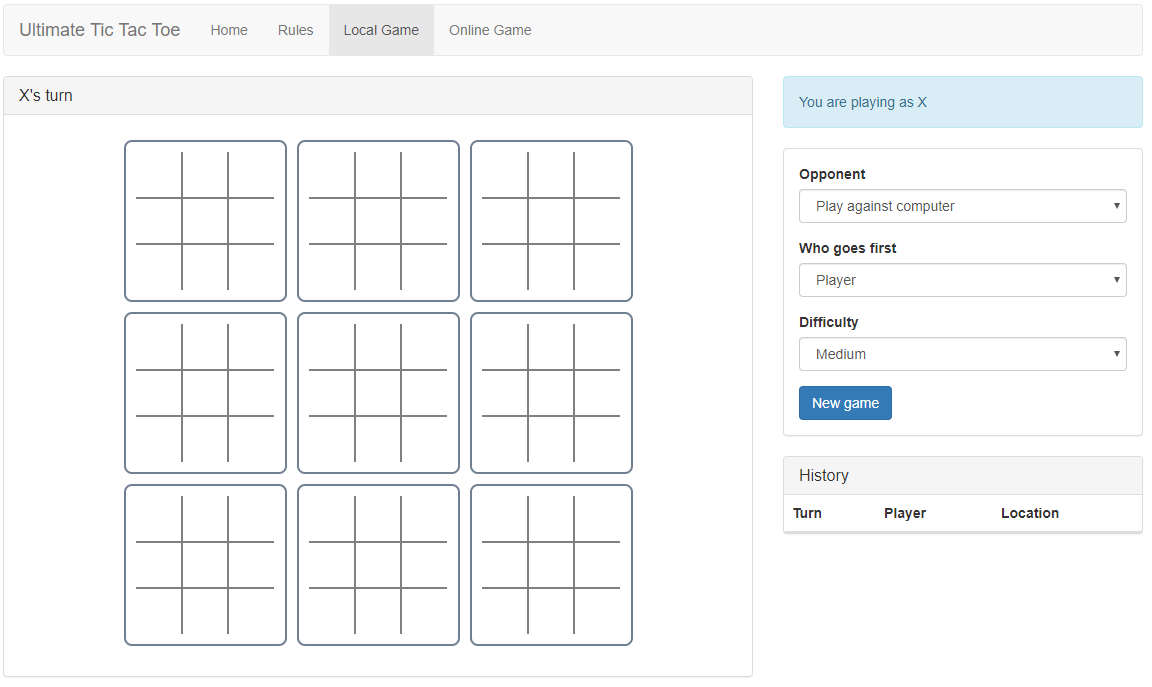
The objective of the game is to win 3 connected local grids before the opponent, similar to in TTT.

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 TTT is not very famous because of its higher complexity and the inability to play it recreationally on paper as with TTT.

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 with the ability to play online, and if no players are found online, the user still has the choice to play the game with AI.

# 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 TTT includes 8 levels of difficulty, while Strategic TTT has only 1 mode of difficulty. Furthermore, Strategic TTT does not allow online multiplayer but does allow multiplayer co-op while Ultimate TTT allows both online multiplayer as well as co-op.

The online multiplayer of Ultimate TTT is achieved by giving a specified link to a friend, which when followed, adds that friend into the room of the user, allowing them to play against each other. I will use a similar method that allows the user to use a code to add a friend to their game since this method is easy, simple to use, and works well.

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

Since Ultimate TTT 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 TTT’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 TTT as opposed to Strategic TTT.

### Key Features:

Ultimate TTT 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 TTT, 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 TTT includes a history bar, which shows all the previous moves that have taken place.

I will attempt to include the first four 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 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.

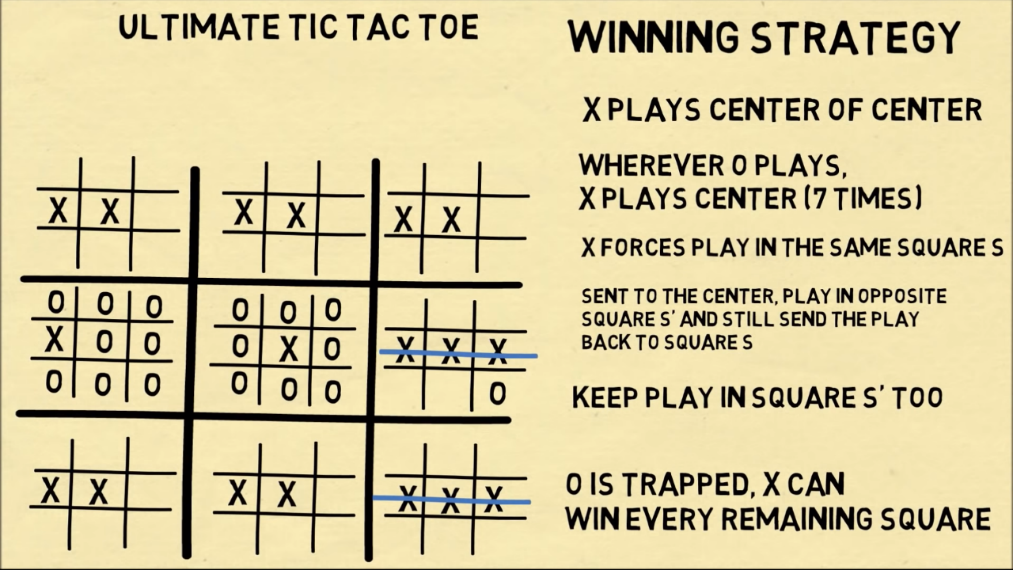
### Rule Variation:

In the games of TTT 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

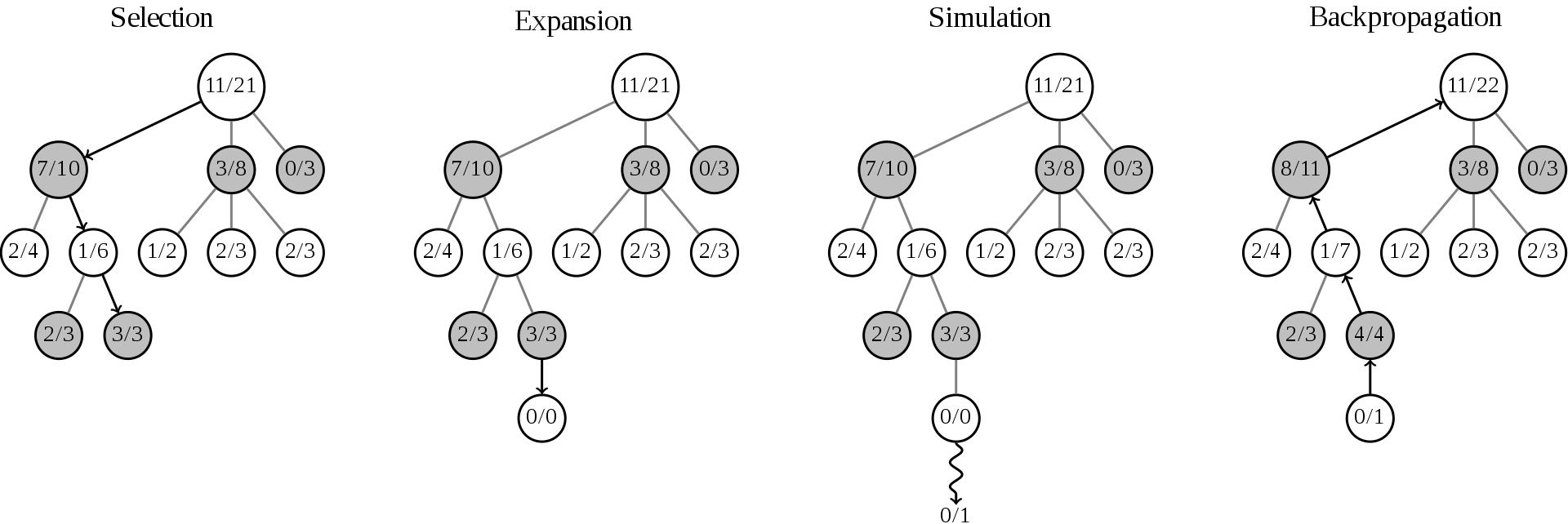
Ultimate TTT is significantly more complex than other variations of TTT.

The difficulty of creating a good AI for Ultimate TTT 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 TTT.

### 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 below), until a leaf node is reached. At the leaf nodes, no simulation step has yet occurred.

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.)

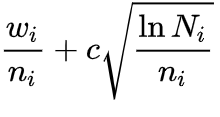
This process continues until either a set amount of time has been reached, or until a given number of iterations have been reached.

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 TTT: 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 TTT 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 TTT 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 game, and turn it off.
3. A ‘Tutorial’ option that takes the player to a tutorial page.
4. A ‘Play Game’ option, which takes the player to the play game mode options page.
5. If time allows, a GUI should be created for the menu as well as the rest of the game.
6. 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.

### 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 highlight clearly the difference in rules between this Ultimate TTT and a minority of other Ultimate TTT games.
3. Should explain the game with an example.
4. Should have a ‘back’ button to allow the player to go back to the main menu page.

### 3-Play Game Options Page:

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

This must include:

1. Options for ‘Online Play’, ‘AI Play’ and ‘Co-Op Play’.
2. There must be a brief explanation of each option.
3. Each option should be clear, and how to select it should be made clear.

### 4-Change AI Difficulty:

In the ‘AI Play’ game mode, this is the ability to make the AI less/more difficult, depending on the user’s preferences.

This must include:

1. The user must be able to change AI difficulty at any point during their turn.
2. There must be at least 3 modes of difficulty, or 4 if time allows, with each mode being significantly more difficult than the previous, but still allowing the user to have fun.
3. The change in difficulty must be produced by either changing the maximum amount of time the Monte Carlo Tree Search algorithm has to produce a result, or by changing the maximum amount of iterations it has to produce a result.

### 5-Find and Connect to Online Opponent:

This must include:

1. The user should be able to connect to a running server.
2. This running server must store other players looking for a match
3. The server must pair the two players up as soon as possible by exchanging IP address and other details between them.
4. Alternatively, if there is enough time, ‘a play with friends’ option may be added that means only players with a 5-character code can join certain matches, which will be stored separately in the server. This would allow only the intended opponent to play with the user.
5. If the connection between the server and user or between the user and opponent is dropped for unknown reasons, there must be a message and error handling to take care of this.
6. If the connection between the two players is dropped because one of the players stops the game, then the other player must win.
7. The connections between the user and opponent and user and server must be fast, and data sent and received must not lag.
8. The server should be able to handle many requests from different players at once, and deal with them all systematically and quickly.

### 6-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 moves of the player and opponent should be created.

### 7-Co-op Opponent:

This is a varied mode of play in which, instead of playing against an opposition on separate devices, the same program on the same device houses 2 players playing against each other.

This must include:

1. A turn-based system that stores both player’s moves in separate data structures.
2. Text should be displayed on screen showing which player’s turn it is.

### 8-Display Game:

Before and after every move, the game board as well as non-game options 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 symbol of the last move made by the opposing player should be highlighted on the game board.
3. The boundaries of the local grid (or grids) that the current player is allowed to play in must be highlighted to prevent confusion.
4. The number of the current player (player 1 or player 2 or AI Player) must be shown.
5. A ‘back’ option must be available at every point in the game, allowing the player to exit the current game and move back to the previous screen.
6. A ‘quit’ option must be available at every point in the game, allowing the player to exit the application entirely.

### 9-User Make Move:

The user is given an intuitive and simple method to input their symbol onto the global grid, where desired.

This must include:

1. 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. The user must then be asked for a confirmation of their move. An intuitive way to achieve this is by asking for a second input. If the second input is the number ‘0’, the move is cancelled and the first move input is requested from the user again.
2. Once the move has been input, 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.
3. Once a valid move is input, it must change the data structure of the current game state accordingly.

### 10-AI Analyse and Make Move:

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

The specific objectives are mentioned below.

### 11-Validate Movement:

Once a move is entered, the current game state data structure must be checked to see it is legal.

This must be done by:

1. Checking that a symbol is not already present in the area that is specified.
2. Checking that the area specified is not in part of the grid which is not currently allowed. I.e. if the current player is only allowed to play in the top right local grid, they cannot place their symbol on the bottom left local grid.
3. If any of these checks return True, the user is given an ‘invalid move’ message, and asked for their move input again.

## 12-Time Out:

1. A timer variable should be counting down the time during multiplayer games.
2. If the player has not made their move for more than 1 minute, the opponent will win automatically.
3. In AI mode, if the player has closed their game, the game state must be saved in a database.

### 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 game state data is replaced with a large X or O, depending on the player, to mark that it has been won.
3. If a win is spotted on the global grid, it must stop the game, and return if the game has been won, lost, or drawn by the user.

### 13-End Screen:

Shows who won, or shows a draw, a ‘quit’ button, a ‘back’ button and a ‘play again’ options are present.

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. A ‘quit’ option to exit the game application.
3. A ‘back’ option to return back to the game options menu.
4. A ‘play again’ option to play the game again in the same game mode.

## AI Data Flow Diagram and Objectives:

### 14-Initialise Game Tree

1. The data structure that stores the game tree must be created.
2. This data structure must be able to store the game state information, as well as the value of wins compared to total simulations, and the UCT value calculated for that node.
3. The current game state must be added to the game tree as the root node.

### 15-Select Node

1. The UCT value must be calculated for every node using information in the game tree data structure.
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), recursively, until a leaf node is reached.

### 16-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.

### 17-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 1/1, and if the end state is a draw, the value must become 0.5/1

### 18-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 added to its parent, and then to the parent’s parent, and so on, until the root node is reached.

### 19-Make Move

1. The selection, expansion, simulation, and back propagation steps must be repeated until a given amount of time runs out. (The amount of time is different depending on the difficulty)
2. Once the time has 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.

### 20-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.

### 21-C Constant Selection

1. The C constant of the UCT value must be determined through testing, and must be different for each difficulty mode for efficient play.

# 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

# 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

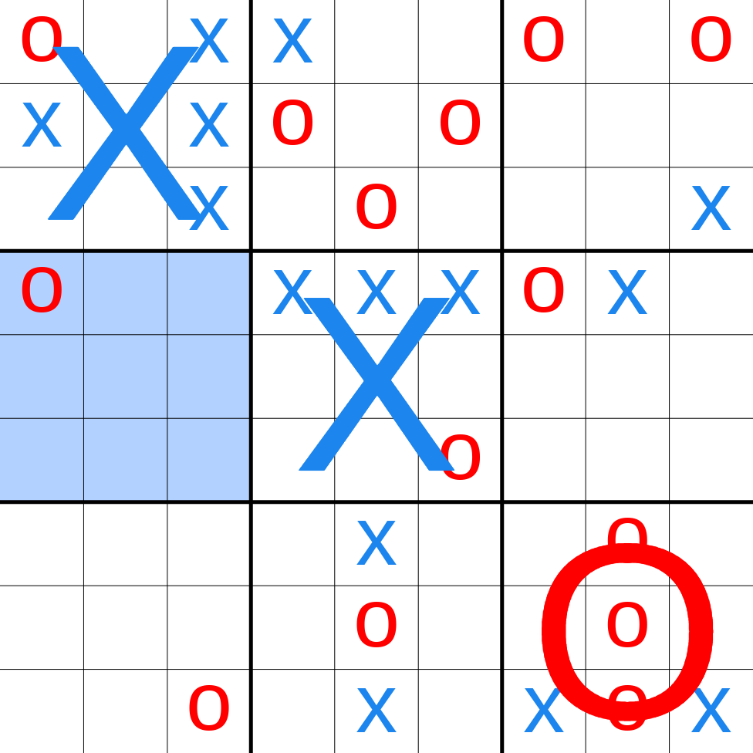
My project is an AI for the game Ultimate Tic Tac Toe, in which the ‘Monte Carlo Tree Search’ algorithm is used by the AI player to make its moves.

# 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.

Definitions:



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

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

# Game Rules:

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.

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

If a player is directed to a local grid that is full, or that has already been won/lost, the player may place their symbol anywhere 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.

# Key Data Structures:

## 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 that local grid.

* Each grid is represented by the position of symbols in the corresponding 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 game state in the picture above, 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’]

## Node.parent

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

Node.parent + legal move = Node

## Node.children

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

Node + legal move = child\_node

## Node.value

Node.value is a tuple containing 2 integers.

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

The first integer is the number of times the node has won in these simulations.

## 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.

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’]

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 previous 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 all of the child nodes.

## 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 and Node.

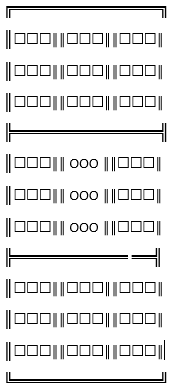
At the root, Node.depth = 0, children of the root have Node.depth = 1, children of these have Node.depth = 2 and so on. By default, Node.depth = Node.parent.depth + 1 and root.depth = 0.

## Game Tree:

The above variables in the Node class form a game tree which connects each Node with its parent and children.

# Game I/O

## Node Display



The Node.state is displayed in the format shown on the left.

It has the following advantageous characteristics:

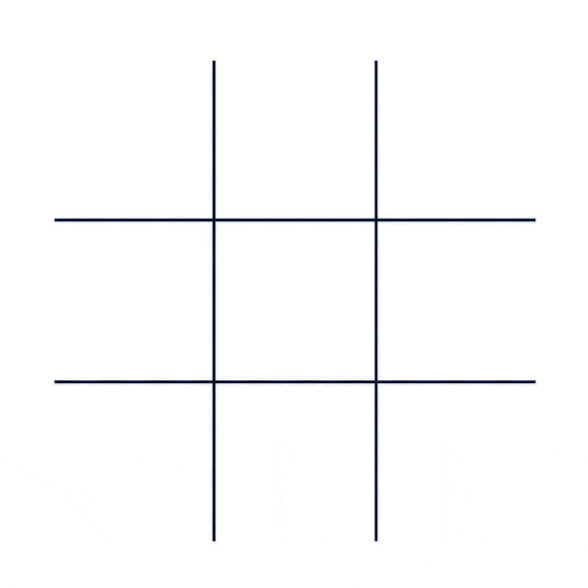
* The 9 local grids are clearly separated by borders.
* All empty spaces are replaced with a ‘☐’ symbol to ensure they can be clearly seen.
* All won/lost grids are replaced by 9 symbols instead of one large

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.

## 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:



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 to the position of the symbol in the local grid.

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

Then a location 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 9, 9.

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 (Further explained in Key Functions 🡪 input\_convertor). For the keypad tuple of 9,9 the new tuple would be 2, 2.

# Key Functions

## Node Class:

|  |  |  |  |
| --- | --- | --- | --- |
| Function Name | Parameters | Return Types | Purpose |
| \_\_init\_\_ | state: list,  parent: Node instance, children: list,  value: tuple,  prev\_move: tuple, current\_player: integer,  depth: integer | None | Initialises all the variables associated with the Node instance.  Further details in Key Data Structures |
| display\_node | None | None | Prints the node.state in the format demonstrated in Game UI → Node Display |

## Global Functions:

|  |  |  |  |
| --- | --- | --- | --- |
| Function Name | Parameters | Return Types | Purpose |
| input\_convertor | coordinate: string | acc\_coordinate: tuple | The coordinate parameter is a string in the format shown in ‘Game I/O 🡪 Node Input’.  This is converted into a tuple which corresponds to the intended Node.state and is returned.  Further explained in Game UI → Function: input\_convertor |
| Opposite | dig: integer | Integer | Gives the opposite of the input integer, returning 1 for 0 and 0 for 1. |
| 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 the Node.state has ended and whether the game has been won/lost or drawn.  Further explained in Key Functions → Function check\_win |

## Function: input\_convertor

def input\_convertor(coordinate):  
 *"""Coordinate: string"""* conv = [7, 8, 9, 4, 5, 6, 1, 2, 3]  
 x, y = int(coordinate[0]), int(coordinate[1])  
 acc\_coordinate = (conv.index(x), conv.index(y))  
 return acc\_coordinate

## Function: check\_move

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

## Function check\_win

Checks if a terminal state 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.

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.

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

# Monte Carlo Tree Search 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

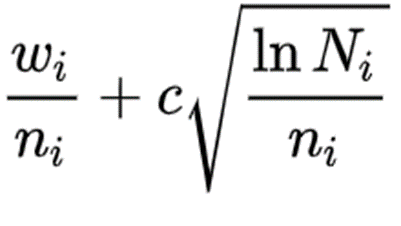
Creates and returns the child node of a parent, such that Node + move = Child Node

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 |
| start\_state = [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '] | Stores the initial state of the game where the global grid is empty, in the format ref |
| self.starting\_player\_num=1 | Stores the value of the player which will take the first turn in the game.  Human=0, AI=1  By default, AI goes first, so this is set to 1. |
| 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 by default 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 turn by the AI or Human in the format seen in ref |
| self.mont = MonteCarlo(grid=start\_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.turn\_count=0 | Stores the turn number.  Increments by 1 after every turn. |

## Game.handle\_input()

Takes the parameter coordinate which is a 2 digit coordinate of where the human player would like to place their symbol on their turn, in the format further explained in ref.

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

def handle\_input(self, coordinate):  
 if len(coordinate) != 2:  
 return False  
  
 for val in coordinate:  
 if val not in ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']:  
 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 not len(self.game\_node.state[self.game\_node.prev\_move[1]]) == 1 \  
 and not a == self.game\_node.prev\_move[1]:  
 return False

## Game.endgame()

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

def endgame(self):  
 if self.result == (1, 1):  
 print("Computer Won!")  
 elif self.result == (0, 1):  
 print("Human Won!")  
 else:  
 print("Draw!")

## Game.run()

The game system for Ultimate Tic Tac Toe.

def run(self):  
 self.game\_node.display\_node()  
 while self.result is None:  
  
 if self.turn == 1:  
 print("AI Turn")  
  
 self.game\_node = self.mont.Monte\_Carlo()

self.prev\_move = self.game\_node.prev\_move  
  
 if self.turn == 0:  
 print("Human Turn")  
 human\_move = input('Coordinates: ')  
 if human\_move == '000':  
 print('Game Aborted')  
 quit()  
  
 while self.handle\_input(human\_move) is False:  
 print("Invalid Move!")  
 human\_move = input('Coordinates: ')  
 if human\_move == '000':  
 print('Game Aborted')  
 quit()  
  
 a, b = input\_convertor(human\_move, reverse=False)  
  
 self.game\_node.state[a] = self.game\_node.state[a][:b] + 'X' + self.game\_node.state[a][b + 1:]  
 self.prev\_move = (a, b)  
  
 self.turn\_count += 1  
  
 self.result = check\_win(self.game\_node.state)  
  
 self.game\_node.display\_node()  
  
 print(input\_convertor(self.prev\_move, reverse=True))  
  
 self.mont.\_\_init\_\_(self.game\_node.state, prev\_move=self.prev\_move)  
  
 self.turn = opposite(self.turn)  
 self.endgame()

Where it says ref- provide ref

Pseudo? How much to explain keeping analysis in mind?

After MCTS class, do Game class.

Testing

Next steps:

Look at Game class, make it not bad

Node class functions explain

Algorithm explain

Algorithm Pseudocode

Rest of functions

Include the pruning and difficulty modes as well as the local Multiplayer stuff in code and finalise code.

Rest of design

Do testing

Clean off analysis

Clean off design

Clean off testing

Say exactly how the UI should be and justify it completely.

Explain clearly the menus and pages designs.

Chart of menus.

Your design may be moderated so needs to be printable.

All screenshots must be perfectly readable.