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.

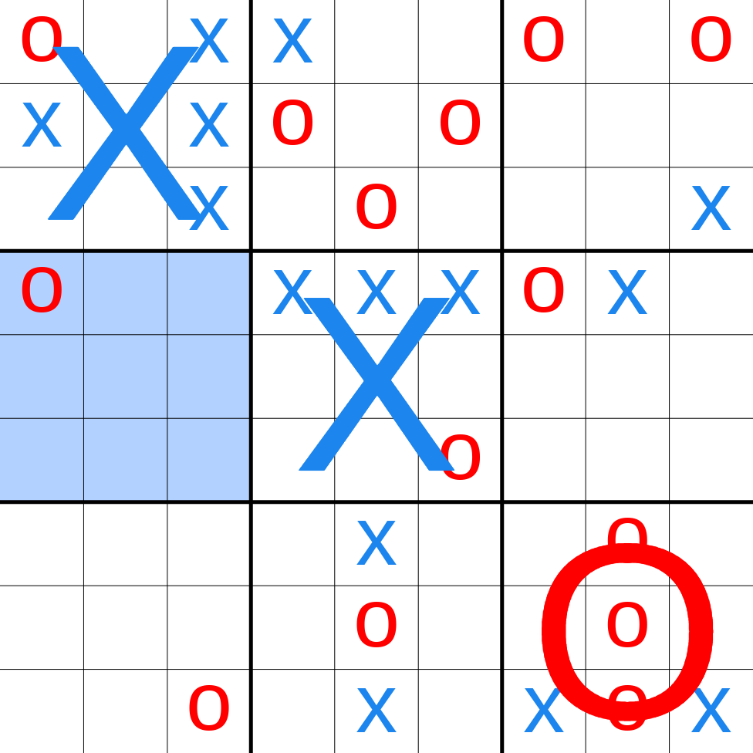
# Project Technology

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.

# 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 a 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, 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.

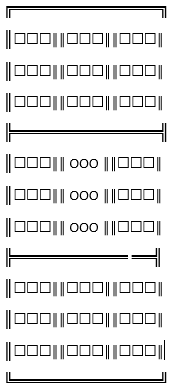
At the root, Node.depth = 0, children of the root have Node.depth = 1, children of these have Node.depth = 2 and so on. By default, Node.depth = Node.parent.depth + 1 and root.depth = 0.

## 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:

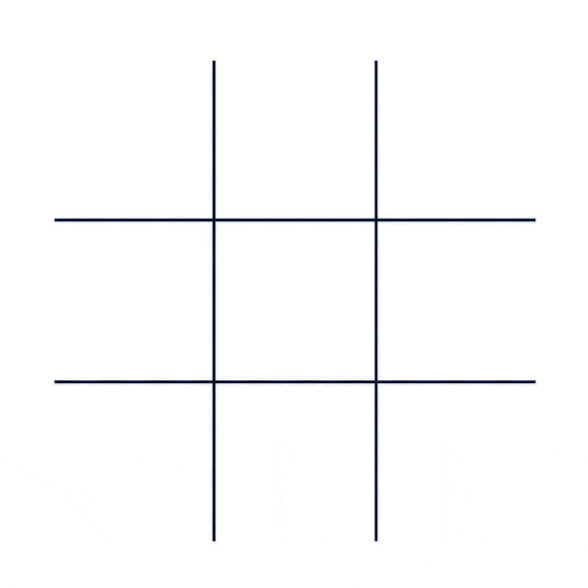
* The 9 local grids are clearly separated by borders.
* All empty spaces are replaced with a ‘☐’ symbol to ensure they can be clearly seen.
* All won/lost grids are replaced by 9 symbols instead of one large

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 keypad format shown in ‘Game I/O 🡪 Node Input’.  This is converted into a tuple in the format  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

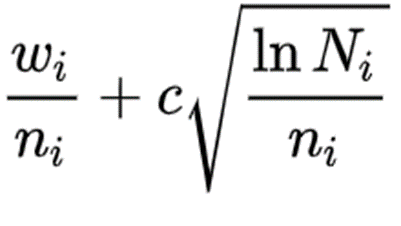
Creates and returns the child node of a parent, such that Node + move = Child Node

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.