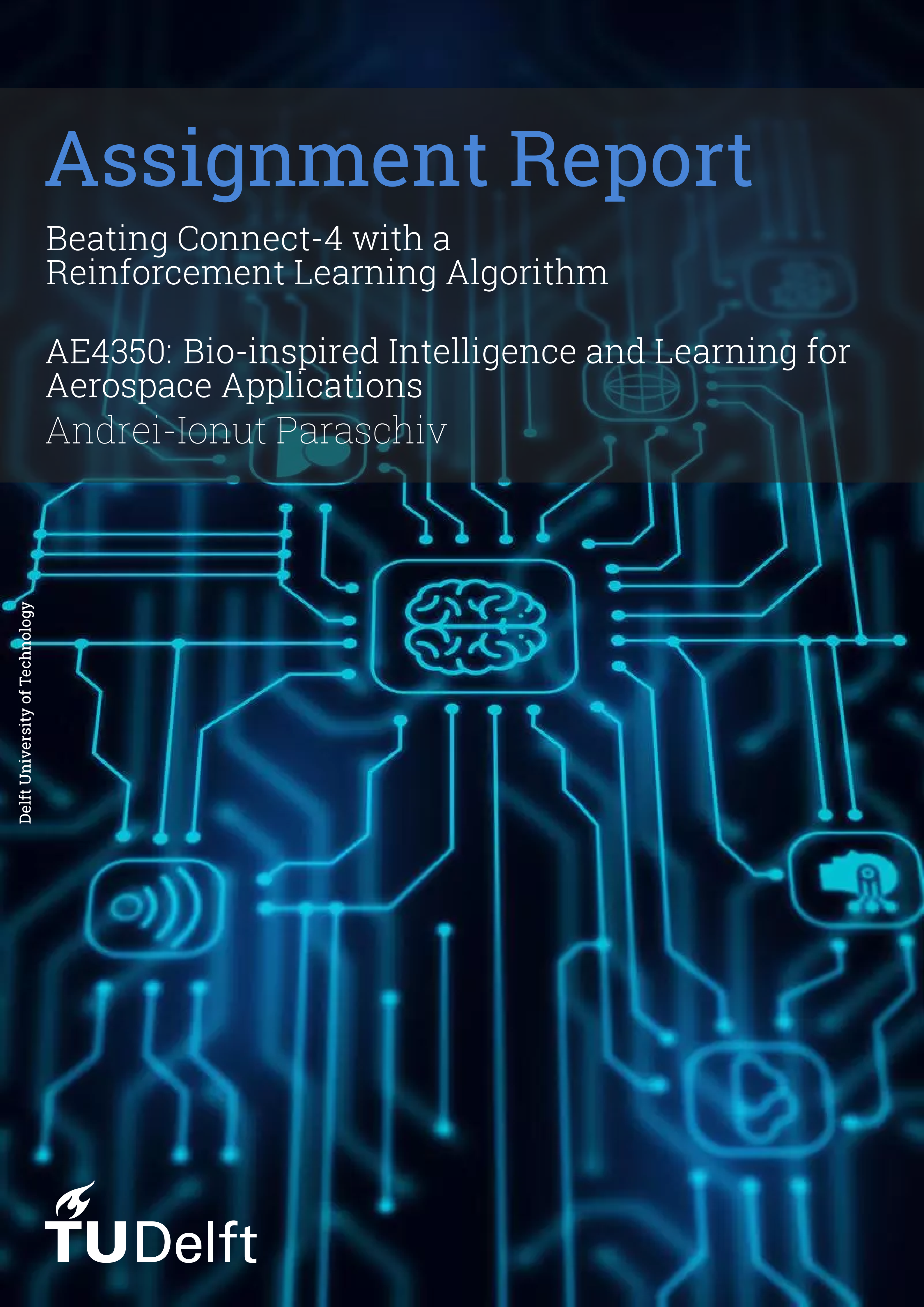

Assignment Report

Beating Connect-4 with a Reinforcement Learning Algorithm

AE4350: Bio-inspired Intelligence and Learning for Aerospace Applications

Andrei-Ionut Paraschiv

Delft University of Technology

Assignment Report

Beating Connect-4 with a Reinforcement Learning Algorithm

by

Andrei-Ionut Paraschiv

Student Name	Student Number
Andrei-Ionut Paraschiv	5955556

Instructors:	Dr. G.C.H.E. de Croon
	Dr.ir. E. van Kampen
Faculty:	Faculty of Aerospace Engineering, Delft

Cover:	AI that approximates the human brain - The Hindu BusinessLine (Rotated)
Style:	TU Delft Report Style, with modifications by Daan Zwaneveld

TU Delft

Contents

Nomenclature

Abbreviations

Abbreviation	Definition
AI	Artificial Intelligence
MCTS	Monte Carlo Tree Search
RL	Reinforcement Learning
UCT	Upper Confidence Bound for Trees

Symbols

Symbol	Definition	Unit
C	Exploration coefficient	[-]
i	Node index	[-]
n	Node	[-]
N	Number of wins made from the node	[-]
p	Node's parent index	[-]
Q	Number of simulations of the node	[-]
UCT	Upper confidence bound for trees	[-]

1 Introduction

The aim of this project is to create an artificial intelligence (AI) agent that can utilize the Monte Carlo Tree Search (MCTS) method to play and win a game of Connect-4. This popular two-player strategy game has an extensive game tree and complex potential game states, which makes it a difficult task for artificial intelligence to overcome. A thoughtful approach to decision-making is necessary to develop an AI that will perform well in Connect-4, and here is where MCTS comes into play.

A strong search technique that is frequently used in agents made for games is the Monte Carlo Tree Search method. Simulating several game scenarios enables the AI to make well-informed decisions by choosing actions that maximize the probability of winning.

While not fundamentally a reinforcement learning (RL) technique, MCTS makes good use of its capacity to simulate a wide range of possible outcomes and systematically examine the game tree to address the problem of decision-making in complex contexts such as Connect-4. Since MCTS may function well with less previous knowledge than many RL algorithms, which require substantial training on vast data sets, it is especially helpful when training time or computer resources are restricted. Furthermore, the way that MCTS strikes a balance between exploration and exploitation is similar to fundamental RL ideas, providing a sound and useful method of solving this game.

1.1. Rules of Connect-4

Two players compete in the strategy game Connect-4 on a vertical grid with seven columns and six rows. A set of twenty-one discs, usually in red and yellow, is provided to each player. The first player to align four of their discs in succession, either horizontally, vertically, or diagonally, is the winner of the game.

One disc at a time, players put theirs into one of the seven columns every turn. A disc that is dropped into a column descends to the lowest point in that column that is accessible. Players take turns to play the game, and every move they make is calculated since they have to stop their opponent from connecting four while still trying to align their own four discs.

The game can end in one of three possible outcomes:

1. **Win:** The first player to connect four of their discs in a row, column, or diagonal wins the game.
2. **Loss:** The second player, the one that did not connect four of his discs, becomes the loser if the first player connects four of his own.
3. **Draw:** If the grid is filled with discs and no player has connected four of their own, the game ends in a draw.

AI researchers frequently study Connect-4 because it is a well-known example of a perfect information game, in which both players always know the whole state of the game.

1.2. The Monte Carlo Tree Search (MCTS)

As mentioned before, a popular heuristic search strategy for making decisions in games like Connect-4 is MCTS. In situations when there is a large search space, where standard techniques may become computationally impractical, MCTS is especially useful. In [1], it is stated that 'It combines the generality of random simulation with the precision of tree search.'

MCTS simulates several random game plays, or "rollouts," from the beginning of the game to its end. To systematically investigate potential actions, it blends these simulations with a tree structure. Usually, there are four primary steps in the MCTS algorithm:

1. **Selection:** Starting from the root node (the current game state), the algorithm selects a child node based on a balance between *exploration* (choosing less-explored moves) and *exploitation* (favoring well-performing moves).
2. **Expansion:** If the selected node represents a non-terminal state (the game has not ended), one or more child nodes are created that represent possible moves that can be made from that state.
3. **Simulation** or **Rollout:** From one of the child nodes, the algorithm simulates a random game and shows the winner.
4. **Backpropagation:** The results of the simulation are propagated back up the tree, updating the nodes along the path with information about the outcome. Nodes that lead to wins are rewarded, and those that lead to losses are penalized.

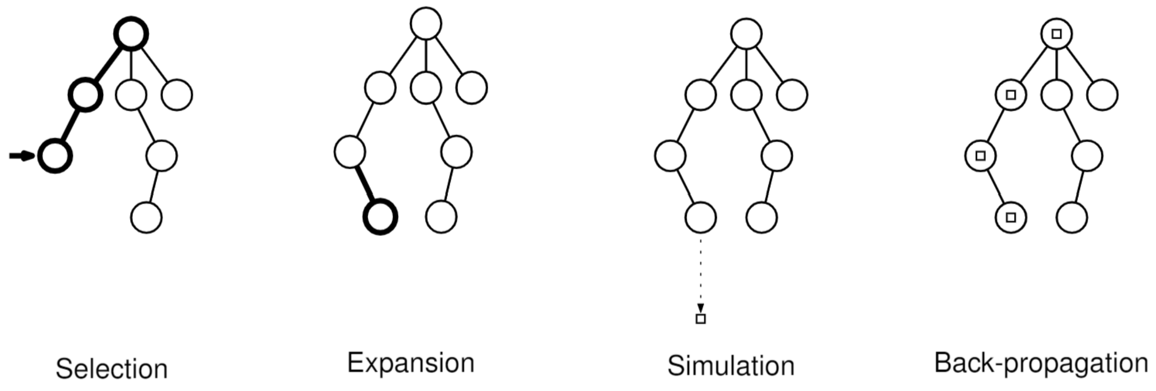

Figure 1.1: The four steps of the MCTS method [3]

The tree fills up with nodes that indicate the possible results of various moves after going through these steps numerous times. The move that has demonstrated the most potential during the search process is then chosen by the algorithm as the move that corresponds to the child node with the highest win ratio or the greatest number of simulations.

Now, as the basic rules of Connect-4 and the fundamentals of the Monte Carlo Tree Search method were explained, the main code and the implementation of the algorithm will be presented in chapter 2.

2 Implementation Overview

The code was written with one goal in mind - to choose the best action based on based on heuristics. Therefore, it wants to find a shortcut to win the Connect-4 game by reaching an optimal result given a time constraint.

The basic idea is to simulate multiple random games from one given state of the game board and find out (based on probability) which move is better.

The agent, then, will perform the move that leads to the state which gives the best chance of winning.

2.1. The Connect-4 Game

The main game of Connect-4 and its rules are implemented in a separate file called **ConnectCode.py** which contains a class that has the same name.

This class starts with the game initialization: it sets up the board (filled with zeroes representing empty cells), keeps track of whose turn it is, as well as stores the last move made (initially an empty list), and tracks the highest empty row in each column.

Next, there is the *move* function which starts by updating the board based on the move chosen and, then, switches the player, updating the storage for the last move, and, finally, adjusts the height of the column.

There are also functions to check if there are legal moves left to be made and get a copy of the game-board without making unintentional modifications.

The win condition is verified after every move by verifying the four possible directions: vertical, horizontal, diagonal, and anti-diagonal. After this, it checks if the game is finished either by win or draw (if there are no legal moves left). Afterward, a function shows which player won or shows a draw.

This class ends with the *print* function, which presents the gameboard in its current state to the user by placing it in the terminal.

2.2. Exploitation and Exploration

The balance between exploitation and exploration is a fundamental idea in many AI algorithms, including MCTS. This balance is critical for creating a competitive AI agent in the context of Connect-4.

The technique of selecting actions that have historically produced positive results is known as **exploitation**. In MCTS, this involves choosing strategies that, according to previous simulations, have a greater win ratio. The AI can successfully rely on tactics that have proven successful in previous iterations by utilizing knowledge that already exists.

On the other hand, **exploration** involves experimenting with less certain or untested actions to identify possibly better approaches. Exploration is essential in MCTS to make sure the agent doesn't overlook the best options because they have not been thoroughly explored. The exploration phase keeps the AI from being overly fixated on a small number of tactics, which could cause it to become predictable or overlook more effective but less obvious actions.

To balance both exploitation and exploration, the Upper Confidence Bound for Trees (UCT) will be used. This parameter will be equivalent to the value of the node in question. UCT is calculated using the following equation:

$$UCT\,(n_i) = \frac{Q}{N} + C \cdot \sqrt{\frac{\log N_p}{N}} \tag{2.1}$$

where n_i represents the node for which UCT is calculated, Q is the number of wins the agent simulated out of the given state through the node in question, N is the number of times the node was explored (number of simulations), C is the exploration coefficient, and N_p is the number of simulations of the parent node.

The uncertainty of some moves (exploration) and the success rate of moves (exploitation) are given a comparable weight in the UCT formula, which directs the selection of nodes in the search tree.

The value of the exploration coefficient was chosen to be $\sqrt{2}$ which is the theoretical value of this parameter as found on [4]. This is considered to be the optimal value for many scenarios because it allows the algorithm to explore enough without skipping good strategies.

2.3. Monte Carlo Tree Search Algorithm

The implementation of the MCTS algorithm is done inside of the **mcts.py** file which contains two classes: *node* and *MCTS*.

The *Node* class represents the individual states (or nodes) within the search tree. They all contain several attributes: *move*, *parent*, *N*, *Q*, *children*, and *outcome*.

The *Node* class contains two functions: adding all children of the current node and calculating the value of the node using Equation 2.1.

The *MCTS* class is responsible for defining the algorithm's phases, conducting the main search, and determining the best possible move from a given state.

It starts by initializing some parameters: an unaltered copy of the initial game state, the current game state (the root node), and some metrics to track the performance of the algorithm (the run-time, the node count, and the number of simulations that were performed).

The four phases of the MCTS algorithm will be presented in the subsequent parts of this section.

The *search* function combines all four phases of the algorithm and repeatedly performs each one within a time limit set by the user. It also tracks the number of simulations and the total time spent in the current search.

Afterward, a function selects the best move (also the move that will be performed) by selecting the child node with the highest visit count.

There is also a function that moves the MCTS tree forward to account for the actual move made in the game. If the move exists among the root's children, the tree will move to that child node. In the opposite case, it will reset the root.

Finally, a function is used to return the number of simulations and the total time taken for the search process.

2.3.1. Selection Phase

This phase starts at the root node and is responsible for selecting a child node based on the highest UCT value. This process continues until a node with no children is reached. If possible, this function also uses the expansion phase to expand the selected node and, then, chooses a random child for the simulation (roll-out).

2.3.2. Expansion Phase

This phase expands a node by generating all possible (and legal) moves from the current state. These moves are added as child nodes to the selected parent node. However, this process is ultimately skipped if the game is already over in the current state.

2.3.3. Simulation Phase

This phase starts a random simulation (chooses random moves until the game ends) from the current state. The outcome of this roll-out is, then, returned, indicating the result of the game.

2.3.4. Back-propagation Phase

This phase is responsible for updating the tree backward (from the selected node to the root node). It updates the visit count N and the cumulative score Q (win count) based on the outcome of the simulation. It is also responsible for alternating the reward between nodes to account for the opponent's perspective.

2.4. Running a Game

The last part of the implementation is the actual game. There are two files for this: one that shows a game between the user (a human player) and the AI agent and one that shows a game between two AI agents.

2.4.1. Human versus Agent

It starts by initializing a new instance of the Connect-4 game (with an empty board) and the MCTS agent.

The game is included in a loop that goes on until the game is over. There are two turns that repeat until the end: the human's turn (first move) and the agent's turn (second move).

The human's turn starts by displaying the gameboard, after which the user types the move, which is then validated by the illegal move check. Afterward, the state is updated for the agent's turn.

On the agent's turn, the MCTS search process is started by allocating the maximum time it is allowed to perform the search. The statistics are displayed and the best move is selected and performed. Afterward, the state is updated again.

Once the while loop is over, the game concludes and the winner is announced. The win condition verification is performed after every move.

2.4.2. Agent versus Agent

The game between two agents is coded similarly to the one between the user and the agent, but it has a few differences.

To allow for better statistics collection and validity, the agents are forced to play multiple games with the same settings. What is more, the statistics are printed and returned to a separate Excel sheet for easier post-processing. The data recorded is the moves chosen by the agents, the number of simulations performed to choose each move, and the time taken for every move.

3 Results and Analysis

Several games were played to allow for a thorough verification of the implementation of the algorithm. For better reporting, this chapter will be split into three separate sections:

1. **Human versus Agent Games:** three games were played against the agent to check its capabilities
2. **Agent versus Agent Games - Same Times:** several games were simulated with the same time allocation between the agents
3. **Agent versus Agent Games - Different Times:** several games were simulated with different time allocation between the agents

3.1. Human versus Agent Games

For this part, the user played two games against the MCTS agent. The agent was set up to perform the search within five seconds. The games can be seen in Figure 3.1 and Figure 3.2.

In these figures, each game board represents two consecutive turns (the user's turn and the agent's turn). The red disks are used by the human player, while the yellow ones are attributed to the agent. In the end, the winning combination is colored in lime green.

It can be seen that both games were won by the agent in a fairly short period (half or a little over half of the discs were used). The average number of simulations (or roll-outs) performed by the MCTS algorithm during each of its turns is 53 034 for the first game and 71 694 for the second game.

It is worth mentioning that, for the second game, during its third move selection, the agent took a little longer than five seconds (5.0156 seconds), which probably led to the higher simulation count.

After these two games, the agent was tested against another Connect-4 solver available online at [2]. This online solver is based on the Alpha-beta pruning search algorithm. This third game can be seen in Figure 3.3.

Figure 3.3 shows that the online solver won the game (the disks highlighted in cyan) after using almost all the disks. This can be attributed to the fact that given almost infinite resources the Alpha-beta pruning algorithm will perform better than MCTS which was set up to search for only five seconds for every turn. The MCTS agent performed an average of 59 200 simulations every turn.

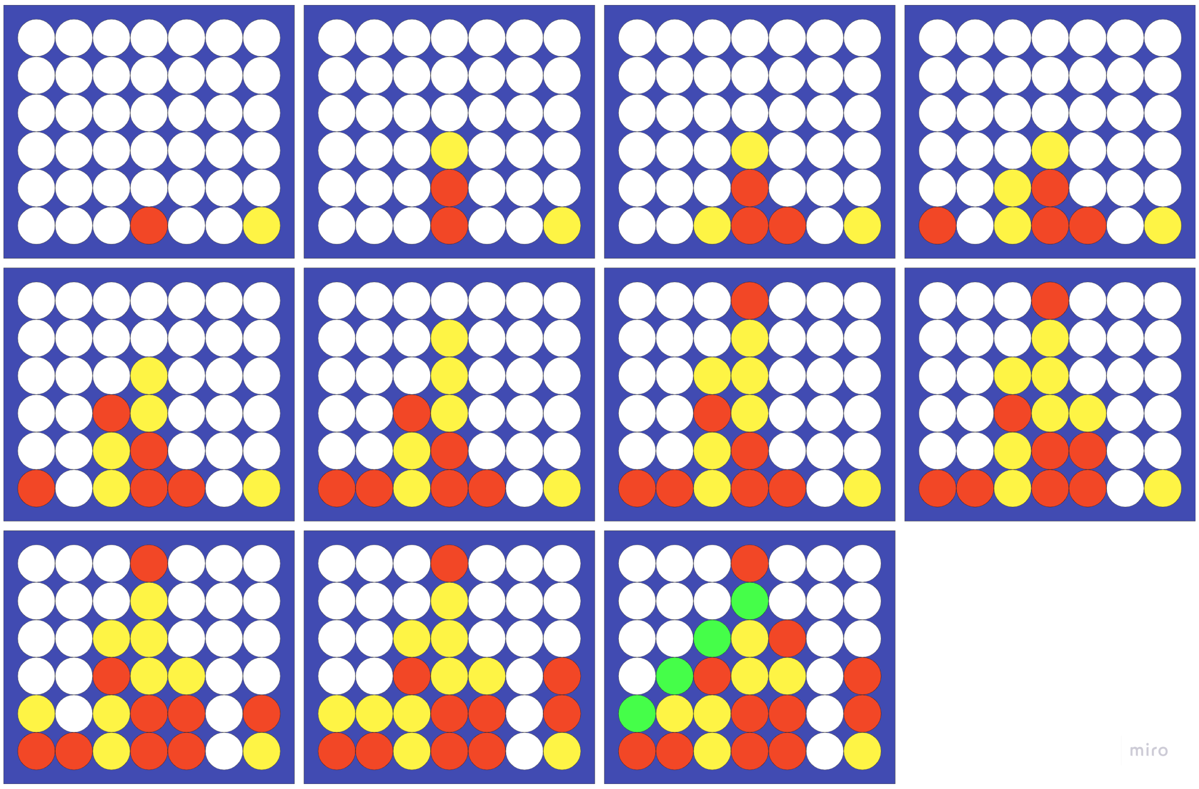

Figure 3.1: All the moves played during the first game

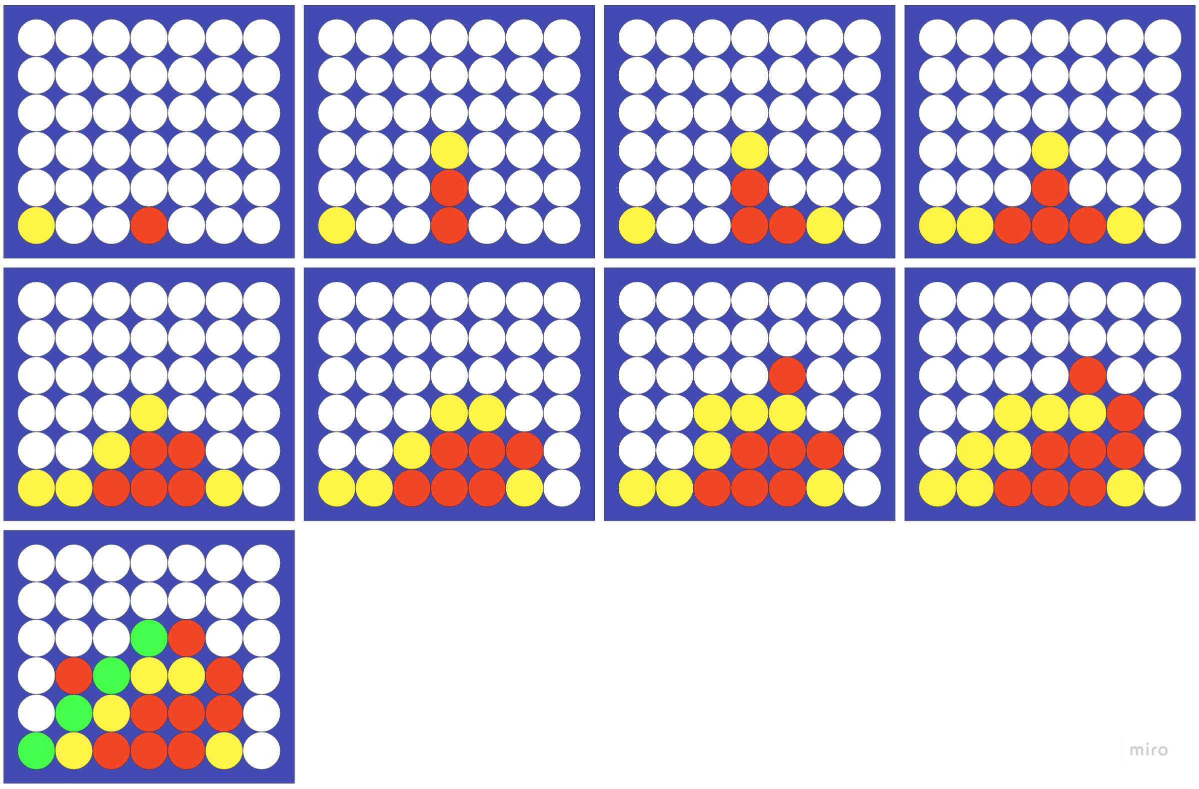

Figure 3.2: All the moves played during the second game

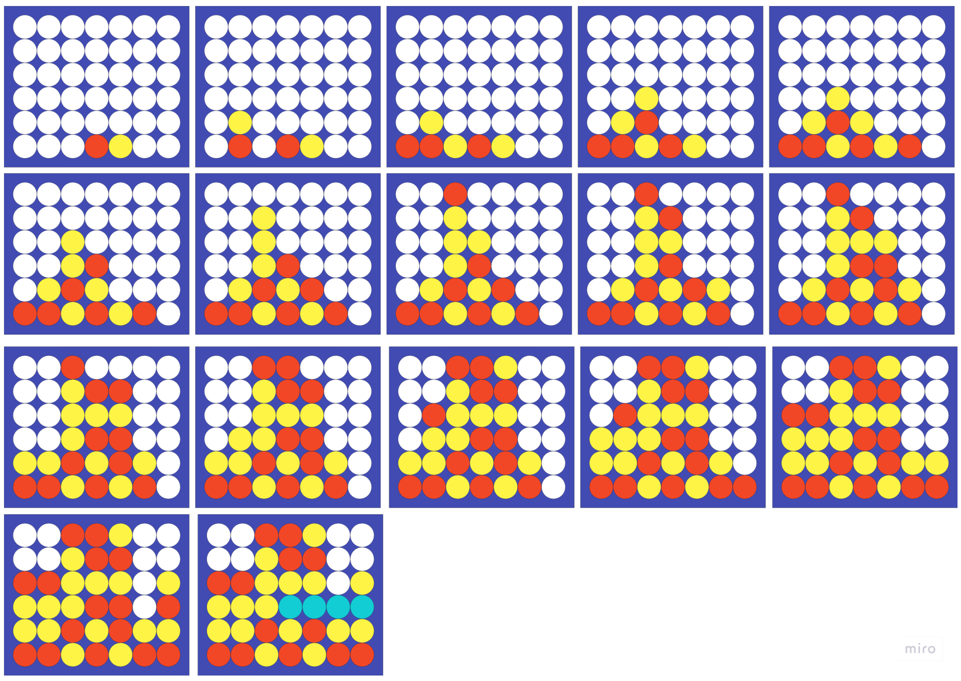

Figure 3.3: All the moves played during the third game

3.2. Agent versus Agent Games - Same Times

In this part, five sets of 100 games each were performed between two identical MCTS agents. The difference between the five sets is the time limit set for the MCTS search. The purpose of these tests is to find out if being the first player plays an important role in winning the game for the two agents (this is the case in a real game). The results of these games can be seen in Table 3.1.

Table 3.1: The results of the games between the two identical agents

Time [seconds]	First Agent		Second Agent	
	Wins	Average Number of Simulations	Wins	Average Number of Simulations
1	52	8 717	48	8 782
5	47	61 560	53	62 028
10	47	103 049	53	104 084
30	43	462 184	57	466 495
60	48	689 843	52	700 477

Table 3.1 shows that the second player is most likely to win the game if they have the same search time. There is only one exception - the first test (for a search time of one second). Disregarding this, it can also be seen that the agent who averaged more simulations than the other one is also the one with the higher win percentage.

3.3. Human versus Agent Games - Different Times

Now, for the last experiment, two final tests were performed. One agent was allowed one second to find its best move, while the other was allowed five seconds. For the first trial (100 games), the first player was given one second, while, for the second trial, the second player had only one second to find their best move. The results of these trials can be seen in Table 3.2.

Table 3.2: The results of the games between the two different agents

Test Number	**First Agent**			**Second Agent**		
	Time [seconds]	**Wins**	**Average Number of Simulations**	**Time [seconds]**	**Wins**	**Average Number of Simulations**
First Test	1	46	12 652	5	54	63 002
Second Test	5	42	56 814	1	58	11 555

The results seen in Table 3.2 are rather interesting. It can be seen that the difference in time does not seem to matter at all. On the contrary, the first agent performs worse when it has more search time available. What is more, it can be observed that the second agent has the higher win percentage in both cases.

4 Conclusion and Discussion

This study's tests provide important insights into how well the Monte Carlo Tree Search (MCTS) algorithm performs in Connect-4 games.

The MCTS agent consistently outperformed the human player in the human versus agent games, winning both of them. This illustrates how the MCTS algorithm works well for quickly assessing possible moves in a constrained amount of time. The average amount of moves in each simulation, ranging from roughly 53 034 to 71 694, shows that even with strict time limits, the algorithm can explore a substantial portion of the game tree. The Alpha-beta pruning-based online solver's victory over the MCTS agent highlights a significant drawback, though: MCTS may not be able to match the depth and accuracy of comprehensive search techniques like Alpha-beta pruning when time is of the essence, particularly when those techniques have access to almost limitless computational resources.

The effect of time on game results was further investigated in the experiments where two identical MCTS agents competed against one another with equal time allocations. The outcomes showed that, with the exception of the scenario with the shortest time limitation (one second), the second participant consistently had a minor advantage in winning. This implies that the second player in MCTS-based games could benefit from knowing the first player's move, enabling them to make better decisions. Furthermore, it appears from the correlation between the number of simulations and the win rate that the agent running more simulations usually gets better outcomes, probably because they are exploring more options in-depth.

Unexpected results were found in the last set of studies, which altered how much time was spent by the two MCTS agents. The agent with more time did not always do better than the one with less time. As a matter of fact, the agent with the lower time allocation won more of the games. Further research into this phenomenon may be needed, maybe involving an examination of the exploration-exploitation balance within the context of MCTS.

These tests' findings show that although MCTS is a strong algorithm that may function effectively in time-constrained settings, its effectiveness is not exclusively reliant on the amount of time available. The order of play and the particular opponent's strategy are two examples of factors that have a big impact on the games' results. The results also highlight how crucial it is to comprehend all aspects of MCTS before applying it to other gaming scenarios or even extending its use to other fields where making decisions in the face of uncertainty is necessary. Optimizing the MCTS algorithm to reduce the second-player advantage seen in the equal-time trials may be the subject of future study. Further research into the decreasing rewards of longer search times may result in an improved approach where time allocation is dynamically modified according to the state of the game or the degree of confidence in the current move evaluation.

In conclusion, this study shows that while MCTS is a powerful tool for game AI, its effectiveness is closely related to both internal algorithmic dynamics and external factors, such as time limitations, which can provide unexpected and informative results.

References

[1] mcts.ai. *Monte Carlo Tree Search*. 2013. URL: `https://mcts.ai/about/index.html` (visited on 08/28/2024).

[2] Pascal Pons. *Connect 4 Solver*. 2019. URL: `https://connect4.gamesolver.org/h` (visited on 08/31/2024).

[3] Andre Santos, Pedro Santos, and Francisco Melo. “Monte Carlo tree search experiments in hearthstone”. In: *2017 IEEE Conference on Computational Intelligence and Games (CIG)* (Aug. 2017), pp. 272–279.

[4] Wikipedia. *Monte Carlo tree search*. 2024. URL: `https://en.wikipedia.org/wiki/Monte_Carlo_tree_search` (visited on 08/31/2024).

A Source Code

Appendix A shows the code used for this assignment. There are 5 files in total:

- **meta.py -** sets up variables for the Connect-4 game and the exploration coefficient for the MCTS algorithm.
- **ConnectCode.py -** the programming of the Connect-4 game.
- **mcts.py -** the coding of the MCTS algorithm.
- **gameHuman.py -** the code that starts the game between an agent and the player.
- **gameAI.py -** the code that starts the game between two agents.

This code can also be found on *GitHub* along with the report using the following link: `https://github.com/AndreiParas/AE4350_Assignment`.

A.1. meta.py

```python
import math

# Misc Parameters

class GameMeta:
    PLAYERS = {'none': 0, 'one': 1, 'two': 2}
    OUTCOMES = {'none': 0, 'one': 1, 'two': 2, 'draw': 3}
    INF = float('inf')
    ROWS = 6
    COLS = 7

class MCTSMeta:
    EXPLORATION = math.sqrt(2) # exploration parameter
```

A.2. ConnectCode.py

```python
from copy import deepcopy
import numpy as np
from meta import GameMeta

# This is the Code for the Connect-4 Game

class ConnectCode:
    def __init__(self): # game initialization - first turn
        self.gameboard = [[0] * GameMeta.COLS for _ in range(GameMeta.ROWS)] # gameboard: '0'
     if no piece, '1' if player one's piece, '2' if players 2 piece
        self.to_play = GameMeta.PLAYERS['one'] # who's playing (first player is player one)
        self.Height = [GameMeta.ROWS - 1] * GameMeta.COLS # height of gameboard
        self.last_played = [] # the last move played - to check for win
    def get_board(self):
        return deepcopy(self.gameboard)
    def move(self, col):
        self.gameboard[self.Height[col]][col] = self.to_play
        self.last_played = [self.Height[col], col]
        self.Height[col] -= 1
        self.to_play = GameMeta.PLAYERS['two'] if self.to_play == GameMeta.PLAYERS['one'] else GameMeta.PLAYERS['one']
    def get_legal_moves(self): # check for illegal move
        return [col for col in range(GameMeta.COLS) if self.gameboard[0][col] == 0]
    def check_win(self): # check who won
        if len(self.last_played) > 0 and self.check_win_from(self.last_played[0], self.last_played[1]):
            return self.gameboard[self.last_played[0]][self.last_played[1]]
        return 0
    def check_win_from(self, row, col): # check if there is a win based on the last move played
        player = self.gameboard[row][col]
        consecutive = 1
        # Check for vertical win
        tmprow = row
        while tmprow + 1 < GameMeta.ROWS and self.gameboard[tmprow + 1][col] == player:
            consecutive += 1
            tmprow += 1
        tmprow = row
        while tmprow - 1 >= 0 and self.gameboard[tmprow - 1][col] == player:
            consecutive += 1
            tmprow -= 1
        if consecutive >= 4:
            return True
        # Check for horizontal win
        consecutive = 1
        tmpcol = col
        while tmpcol + 1 < GameMeta.COLS and self.gameboard[row][tmpcol + 1] == player:
            consecutive += 1
            tmpcol += 1
        tmpcol = col
        while tmpcol - 1 >= 0 and self.gameboard[row][tmpcol - 1] == player:
            consecutive += 1
            tmpcol -= 1
        if consecutive >= 4:
            return True
        # Check for diagonal win
        consecutive = 1
        tmprow = row
        tmpcol = col
        while tmprow + 1 < GameMeta.ROWS and tmpcol + 1 < GameMeta.COLS and self.gameboard[tmprow + 1][tmpcol + 1] == player:
            consecutive += 1
            tmprow += 1
            tmpcol += 1
        tmprow = row
        tmpcol = col
        while tmprow - 1 >= 0 and tmpcol - 1 >= 0 and self.gameboard[tmprow - 1][tmpcol - 1] == player:
            consecutive += 1
            tmprow -= 1
            tmpcol -= 1
        if consecutive >= 4:
            return True
        # Check for anti-diagonal win
        consecutive = 1
        tmprow = row
        tmpcol = col
        while tmprow + 1 < GameMeta.ROWS and tmpcol - 1 >= 0 and self.gameboard[tmprow + 1][tmpcol - 1] == player:
            consecutive += 1
            tmprow += 1
            tmpcol -= 1
        tmprow = row
        tmpcol = col
        while tmprow - 1 >= 0 and tmpcol + 1 < GameMeta.COLS and self.gameboard[tmprow - 1][
    tmpcol + 1] == player:
            consecutive += 1
            tmprow -= 1
            tmpcol += 1
        if consecutive >= 4:
            return True
        return False
    def game_over(self): # check if the game is done
        return self.check_win() or len(self.get_legal_moves()) == 0
    def get_outcome(self): # get the outcome
        if len(self.get_legal_moves()) == 0 and self.check_win() == 0:
            return GameMeta.OUTCOMES['draw']
        return GameMeta.OUTCOMES['one'] if self.check_win() == GameMeta.PLAYERS['one'] else GameMeta.OUTCOMES['two']
    def print(self): # printing the board
        print('=====================================')
        for row in range(GameMeta.ROWS):
            for col in range(GameMeta.COLS):
                print('|| {} '.format('X' if self.gameboard[row][col] == 1 else 'O' if self.gameboard[row][col] == 2 else ' '), end='')
            print('||')
        print('=====================================')
```

A.3. mcts.py

```python
import random
import time
import math
from copy import deepcopy
from ConnectCode import ConnectCode
from meta import GameMeta, MCTSMeta

# MCTS Algorithm

class Node: # this is the 'node' class
    def __init__(self, move, parent): # initial state
        self.move = move # the move that lead to this state
        self.parent = parent # parent node
        self.N = 0 # number of times the node was visited (initially 0)
        self.Q = 0 # number of won simulations that resulted from this node
        self.children = {} # represents possible moves to be made next (children nodes)
        self.outcome = GameMeta.PLAYERS['none'] # the game outcome for this node
    def add_children(self, children: dict) -> None: # function to add all children
        for child in children:
            self.children[child.move] = child
    def value(self, explore: float = MCTSMeta.EXPLORATION):
        if self.N == 0: # this prioritizes unexplored nodes
            return 0 if explore == 0 else GameMeta.INF
        else:
            return self.Q / self.N + explore * math.sqrt(math.log(self.parent.N) / self.N) # value based on UCT

class MCTS: # this is the main 'Monte Carlo Tree Search' class
    def __init__(self, state=ConnectCode()): # initialization function
        self.root_state = deepcopy(state)
        self.root = Node(None, None)
        self.run_time = 0
        self.node_count = 0
        self.num_rollouts = 0
    def select_node(self) -> tuple: # node selection function
        node = self.root # starts from the root node (which is the last move played)
        state = deepcopy(self.root_state) # takes the state of the board
        while len(node.children) != 0: # selects the node with the highest value (based on UCT)
            children = node.children.values()
            max_value = max(children, key=lambda n: n.value()).value()
            max_nodes = [n for n in children if n.value() == max_value]
            node = random.choice(max_nodes)
            state.move(node.move)
            if node.N == 0: # forces exploration of an unexplored node
```

```
                return node, state
        if self.expand(node, state): # expand the children if possible and choose a randome children
            node = random.choice(list(node.children.values()))
            state.move(node.move)
        return node, state
    def expand(self, parent: Node, state: ConnectCode) -> bool: # node expansion function
        if state.game_over():
            return False
        children = [Node(move, parent) for move in state.get_legal_moves()]
        parent.add_children(children) # just adding the children
        return True
    def roll_out(self, state: ConnectCode) -> int: # roll-out or simulation function
        while not state.game_over(): # random moves until game is over
            state.move(random.choice(state.get_legal_moves()))
        return state.get_outcome()
    def back_propagate(self, node: Node, turn: int, outcome: int) -> None: # backpropagation function
        # For the current player, not the next player
        reward = 0 if outcome == turn else 1 # +1 reward if the game ends on the agents turn
        while node is not None: # updates the tree
            node.N += 1
            node.Q += reward
            node = node.parent
            if outcome == GameMeta.OUTCOMES['draw']:
                reward = 0
            else:
                reward = 1 - reward
    def search(self, time_limit: int): # combination of all 4 phases (selection, expansion, roll-out/simulation, and backpropagation) function
        start_time = time.process_time()
        num_rollouts = 0
        while time.process_time() - start_time < time_limit: # repeats the 4 MCTS steps in the time allocated
            node, state = self.select_node() # selects the node and expands it
            outcome = self.roll_out(state) # gets the outcome of the random simulation
            self.back_propagate(node, state.to_play, outcome) # updates all the parents
            num_rollouts += 1 #
        run_time = time.process_time() - start_time
        self.run_time = run_time # the run time of the MCTS (saved for statistics)
        self.num_rollouts = num_rollouts # the number of MCTS simulated games (saved for statistics)
    def best_move(self): # best move selection function
        if self.root_state.game_over():
            return -1
        max_value = max(self.root.children.values(), key=lambda n: n.N).N
        max_nodes = [n for n in self.root.children.values() if n.N == max_value]
        best_child = random.choice(max_nodes) # makes the move that takes us to the state with the highest 'N' (most visited state)
        return best_child.move
    def move(self, move): # the function that moves the state of the tree to the new based on the move made by the player
        if move in self.root.children:
            self.root_state.move(move)
            self.root = self.root.children[move]
            return
        self.root_state.move(move)
        self.root = Node(None, None)
    def statistics(self) -> tuple: # statistics function
        return self.num_rollouts, self.run_time
```

A.4. gameHuman.py

```
from ConnectCode import ConnectCode
from mcts import MCTS

# This is for Human (p1) versus Agent (p2)

def play():
    state = ConnectCode() # initial new gameboard
```

```python
    mcts = MCTS(state) # initializes the agent
    while not state.game_over(): # runs a whole game
        print("Current state:")
        state.print()
        user_move = int(input("Enter a move: "))
        while user_move not in state.get_legal_moves():
            print("Illegal move")
            user_move = int(input("Enter a move: "))
        state.move(user_move) # changes the state of the gameboard
        mcts.move(user_move) # updates the MCTS tree based on the last move
        state.print()
        if state.game_over():
            print("Player one won!")
            break
        print("Thinking...")
        mcts.search(5) # the time(seconds) allocated to the agent to make its move
        num_rollouts, run_time = mcts.statistics()
        print("Statistics: ", num_rollouts, "rollouts in", run_time, "seconds")
        move = mcts.best_move() # selects the best move
        print("MCTS chose move: ", move)
        state.move(move) # changes the state of the gameboard
        mcts.move(move) # updates the MCTS tree based on the last move
        if state.game_over():
            print("Player two won!")
            break
if __name__ == "__main__":
    play()
```

A.5. gameAI.py

```python
from ConnectCode import ConnectCode
from mcts import MCTS
import pandas as pd

# This is for Agent (p1) versus Agent (p2)

def play():
    aux = 0 # to arrange the excel output
    wins_p1 = 0 # count for number of wins made by Agent 1
    wins_p2 = 0 # count for number of wins made by Agent 1
    for x in range(1): # change to how many games you want to be played
        state = ConnectCode() # initial new gameboard
        mcts = MCTS(state) # initializes the agents
        moves_1 = []
        sims_1 = []
        times_1 = []
        moves_2 = []
        sims_2 = []
        times_2 = []
        while not state.game_over(): # runs a whole game
            print("Current state:")
            state.print()
            print("Thinking...")
            mcts.search(1) # the time(seconds) allocated to agent 1 to make its move
            num_rollouts, run_time = mcts.statistics()
            print("Statistics: ", num_rollouts, "rollouts in", run_time, "seconds")
            move = mcts.best_move() # selects the best move
            print("MCTS(1) chose move: ", move)
            moves_1.append(move)
            sims_1.append(num_rollouts)
            times_1.append(run_time)
            state.move(move)
            mcts.move(move)
            state.print()
            if state.game_over():
                print("MCTS(1) won!")
                wins_p1 = wins_p1 + 1
                break
            print("Thinking...")
            mcts.search(1) # the time(seconds) allocated to agent 2 to make its move
            num_rollouts, run_time = mcts.statistics()
            print("Statistics: ", num_rollouts, "rollouts in", run_time, "seconds")
            move = mcts.best_move() # selects the best move
            print("MCTS(2) chose move: ", move)
            moves_2.append(move)
            sims_2.append(num_rollouts)
            times_2.append(run_time)
            state.move(move)
            mcts.move(move)
            if state.game_over():
                print("MCTS(2) won!")
                wins_p2 = wins_p2 + 1
                break
        # Output to excel
        df_moves_1 = pd.DataFrame(moves_1)
        df_moves_2 = pd.DataFrame(moves_2)
        df_sims_1 = pd.DataFrame(sims_1)
        df_sims_2 = pd.DataFrame(sims_2)
        df_times_1 = pd.DataFrame(times_1)
        df_times_2 = pd.DataFrame(times_2)
        with pd.ExcelWriter('output.xlsx', if_sheet_exists= 'overlay', mode='a') as writer:
            df_moves_1.to_excel(writer, index=False, header=False, startcol=aux)
            df_sims_1.to_excel(writer, index=False, header=False, startcol=aux+1)
            df_times_1.to_excel(writer, index=False, header=False, startcol=aux+2)
            df_moves_2.to_excel(writer, index=False, header=False, startcol=aux+3)
            df_sims_2.to_excel(writer, index=False, header=False, startcol=aux+4)
            df_times_2.to_excel(writer, index=False, header=False, startcol=aux+5)
        aux = aux + 7
        print('Player 1 has won', wins_p1, 'times')
        print('Player 2 has won', wins_p2, 'times')
        print('Games done:', x+1)
if __name__ == "__main__":
    play()
```