

Web-Based AI Game Platform for AI Agents

Final Year Project – 2025

B.Sc. Single Honours in Computer Science and Software Engineering



**Maynooth
University**
National University
of Ireland Maynooth

Department of Computer Science

Maynooth University

Maynooth, Kildare

A thesis submitted in partial fulfilment of the requirements for the B.Sc. Single Honours in Computer Science and Software Engineering.

Project Type: Standard

Supervisor: Dr Edgar Galvin

CONTENTS

| | |
|---|-----|
| List of Figures..... | i |
| Abstract..... | iii |
| Chapter 1: INTRODUCTION..... | 1 |
| Chapter 2: MONTE-CARLO TREE SEARCH..... | 2 |
| 2.1 Monte-Carlo Simulation..... | 2 |
| 2.2 Monte-Carlo Tree Search..... | 3 |
| 2.2.1 Upper Confidence Bound for Trees..... | 3 |
| 2.2.2 Upper Confidence Bounds 1 (UCB1)..... | 4 |
| 2.2.3 UCT Variations..... | 4 |
| 2.3 MCTS in AI..... | 4 |
| 2.4 Training AI Agents..... | 5 |
| Chapter 3: GAMES..... | 6 |
| 3.1 Tic Tac Toe..... | 6 |
| 3.1.1 The Rules and Strategies of Tic Tac Toe..... | 6 |
| 3.2 Connect Four..... | 7 |
| 3.2.1 The Rules and Strategies of Connect Four..... | 7 |
| Chapter 4: IMPLEMENTATION..... | 8 |
| 4.1 Monte-Carlo Tree Search in Tic Tac Toe..... | 8 |
| 4.2 Monte-Carlo Tree Search in Connect Four..... | 10 |
| Chapter 5: TESTING AND RESULTS..... | 11 |
| Chapter 6: CONCLUSION..... | 12 |
| REFERENCES..... | 13 |

LIST OF FIGURES

Figure 1 Four-step Monte-Carlo Tree Search processes

Figure 2 Monte-Carlo Tree Search in Tic Tac Toe

Figure 3 Connect Gameplay

Figure 4 UML Diagram of the Tic Tac Toe backend application

Figure 5 Simulation Test Results

ABSTRACT

The advancement of AI in board games over time has led to an extensive use and study of search-based algorithms such as the Monte-Carlo Tree Search (MCTS). In this paper I discuss and describe how the MCTS algorithm is implemented in AI agent games and competes against them in two well-known board games; namely Tic Tac Toe and Connect Four. By applying MCTS to both games, I assess its ability to strategically explore game states, balance exploitation and exploration and improve decision-making over numerous simulations. The impact of the algorithm is evident in its ability to provide an exceptional challenge to human and AI opponents. Given the differences in the complexity and search space of both games, I evaluate how MCTS adapts to different game complexities, develops move selection and performs when it encounters different simulation constraints.

Furthermore, I explore how increasing the number of simulations per move affects the AI's performance, decision quality, computational efficiency and game outcomes. This study focuses on the effectiveness and efficiency of MCTS as a powerful approach in board games and its limitations when handling deeper search spaces with time and number of simulation constraints.

1. INTRODUCTION

AI has made a significant improvement in developing algorithms that are capable of outperforming human players in various complex board games, notably enhanced through the application of search-based methodologies. MCTS has become one of the leading algorithms within this domain due to its effective balance between exploitation and exploration during decision making processes [4]. Initially made popular by the revolutionary achievement in the game of Go [5], MCTS displayed unrivalled capabilities in handling extensive search spaces with only minimal prior domain-specific knowledge.

MCTS utilises randomly simulated playouts to estimate the value of potential moves while balancing between exploring new moves and exploiting those known to produce a favourable outcome. This algorithm is well-suited for games with large, complex decision trees, where traditional methods such as Minimax can be computationally infeasible or less efficient [2].

The algorithm dynamically adjusts the exploration and exploitation strategies in games like Tic Tac Toe and Connect Four to manage their varied game complexities. Tic Tac Toe presents a simpler context with a more limited state space, making it an ideal environment to analyse fundamental decision-making improvements through MCTS. Connect Four introduces a considerably significant complexity, challenging the algorithm to efficiently explore numerous possibilities and select optimal moves within the given constraints, providing a deeper insight into how MCTS adapts to different game complexities [10].

2. MONTE-CARLO TREE SEARCH

2.1 Monte-Carlo Simulation

Monte-Carlo simulation is a simulation-based search algorithm that predicts the probability of a variety of outcomes based on random actions from the root node. The algorithm can simulate results from basic problems such as flipping of a head-or-tails coin to complex issues such as making a move in a chess game. Simulation in games always begins on the root node s_0 until termination with a use of a fixed simulation policy.

In games, the Monte-Carlo simulation runs a large number of trials and averages the results to estimate the outcome and predict if a move will provide gain or loss, it provides a straightforward approach for approximating root value $Q \pi(s_0, a)$. From the state s , $N(s)$ complete games are played out using the policy π through self-play. The Monte-Carlo Value is calculated as the average outcome of all simulations where the action a was taken in state s ,

$$Q(s, a) = \frac{1}{N(s, a)} \sum_{i=1}^{N(s)} I_i(s, a) z_i$$

Where z_i represents the result of the i -th simulation, $I_i(s, a)$ is an indicator function that returns 1 if the action a was chosen in state s during the i -th simulation, otherwise it returns 0. $N(s, a) = \sum_{i=1}^{N(s)} I_i(s, a)$ sums up the total of the total number of times the action a was chosen in state s .

As much as the Monte-Carlo simulation is used to estimate actions rather than to enhance the simulation policy itself, its fundamentals method and functioning can be enhanced either by successively favouring actions with more potential and tend to be more successful or by removing actions that are prone to less success.

In games with clearly stated evaluation functions such as chess or checkers, truncated Monte-Carlo simulation is more successful. This method involves halting simulations before the game ends and estimating the outcome based on the current game state. It enhances simulation speed and reduces the variance of results. However, in more complex games like Go [4], it is challenging to create an accurate evaluation function; truncating simulations usually introduces greater biases in evaluation than the reduction in variance it might offer. Therefore, in such complex games, it is always better to continue simulations until termination providing more exploration and a deeper understanding of potential outcomes.

2.2 Monte-Carlo Tree Search

MCTS is a heuristic tree search algorithm that uses Monte-Carlo simulation for better decision-making [5]. The Monte-Carlo simulation search algorithm assesses the nodes of a search tree by simulating complete games a root state s_0 until termination using a fixed simulation policy. MCTS is a best-first technique: at each step of simulation, it selects the best child, enabling it to continuously refocus its

attention on the greatest value regions of the state space. As the tree increases in size the values at the nodes increasingly approximate the minimax values (best reward against the best defense), the more the simulations, the better the understanding of the outcomes of different moves for the algorithm. Additionally, as the tree increases in size, the simulation policy gradually approaches an optimal minimax policy (the selection mechanism focuses on the most promising parts of the search space).

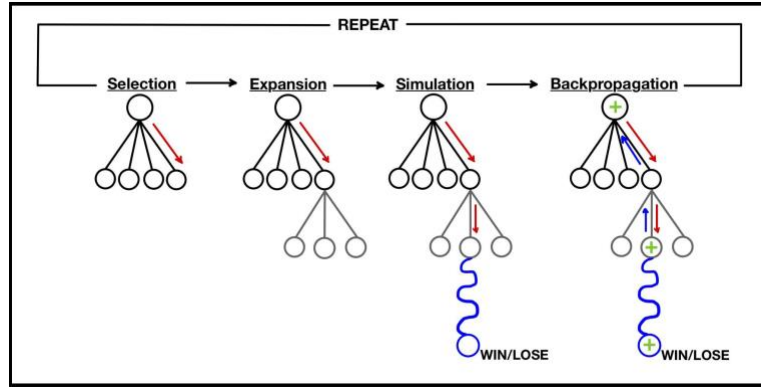


Fig 1. Four-step MCTS process runs in a loop until termination

MCTS combines a four-step process to encourage better choices and moves:

- **Selection:** Starting from the root node (current game state), the algorithm traverses (moves down) the tree and selects the most promising node using the Upper Confidence Bound for Trees (UCT) to find the balance between exploitation and exploration of nodes.
- **Expansion:** If we reach a node that has not been explored, a new child node is added to the tree (new game state), this node may also be explored for a possible move.
- **Simulation:** A random simulation or playout involving making moves at random until a win, draw or loss result is reached begins from the newly expanded node until we reach a termination state. The results are saved as either a reward or loss.
- **Backpropagation:** Each tree node that was traversed is updated with the reward/loss and the number of visits from the simulation. The path that results in a win will receive reward scores and this path will be remembered as one of the winning states.

2.2.1 Upper Confidence Bound for Trees

Upper Confidence Bounds for Trees (UCT) helps find a better simulation score by balancing exploitation of known rewards with the exploration of less or unvisited nodes with the use of Upper Confidence Bounds 1 (UCB1) [6]. The UCT formula balances two key factors:

Exploitation: nodes with high a win rate is chosen hence the MCTS algorithm can explore moves that have been proved successful.

Exploration: nodes with less visits are chosen hence the MCTS algorithm can explore new and potentially promising nodes.

2.2.2 Upper Confidence Bounds 1 (UCB1)

The Upper Confidence Bounds (UCB1) is a standard UCT formula which primarily considers the balance between exploration of unvisited nodes and exploitation of nodes with potential or promising win outcomes without additional complexities. UCB1 does not require immense computational resources hence board games like Tic Tac Toe and Connect4 can utilise it without the worry of computational overhead.

$$UCT = \left(\frac{w_i}{n_i}\right) + C\left(\sqrt{\frac{\ln N_i}{n_i}}\right)$$

Where,

- w_i is the number of wins after the i -th move.
- n_i is the number of simulations after the i -th move.
- N_i is the total number of simulations that have passed through the parent node.
- C represents the exploration parameter, usually $\sqrt{2}$.
- $\frac{w_i}{n_i}$ represents the exploitation parameter (always favours nodes with a high win ratio).
- $C\left(\sqrt{\frac{\ln N_i}{n_i}}\right)$ represents the exploration term (always favours that are yet to be explored).

The formula is then applied to MCTS as a multi-armed bandit, where each move corresponds to an arm of the bandit helping MCTS determine the most promising node (move) to explore further and explore moves that are yet to be explored.

2.2.3 UCT Variations

UCT has other variations that are utilised for different industries such as how AlphaZero neural networks uses Predictor + UCT (PUCT) to guide the MCTS algorithm [7], enhancing decision-making in games like Chess [7] and Go [5] by making use of the algorithm's ability to efficiently navigate large search spaces. The algorithm's ability to learn new strategies through self-play ensured AlphaZero's continuous improvement by playing itself and updating its neural network based on the learned results. The difference with PUCT is that it included the integration of a predictor policy which modifies the standard UCB1 algorithm by adding an action probability policy provided by the predictor into the decision-making.

Rapid Action Value Estimation (RAVE) is another improvement to the MCTS algorithm that is designed to accelerate the learning process in games with large branching factors. It mainly uses the All Moves As First (AMAF) heuristic search to enhance the efficiency of the algorithm [5]. Games like Go that involve more complex strategic and tactical play, this approach is useful because certain moves are more applicable across different game states.

2.3 MCTS in AI

The advancement from Monte-Carlo simulations has led to MCTS to be widely used in the real-world. Industries such as Robotics and Autonomous Systems [8], Board Games [9] & Game Theory [5] and AI & Machine Learning [5] have made use of its decision-making capacity. One of the most notable uses of

the algorithm has been as an AI program for the Asian board game Go. With 10^{170} moves and up to 361 legal moves, the 19x19 square grid game has a huge search space such that other algorithms such as alpha-beta pruning find it hard to optimise it successfully. MCTS with UCT, unlike the aforementioned algorithm can handle large tree sizes better through sampling and progressive expanding of tree nodes, random simulations balanced with expiration and exploitation make it easier to handle uncertainty and the focus on the most promising nodes make the algorithm computationally efficient and support parallelisation.

These factors, among others, have made MCTS be the ideal algorithm to be implemented in more board games including Chess, Tic Tac Toe and Connect Four and their variations.

2.4 Training AI Agents

MCTS has had a huge impact on artificial intelligence especially in the training of AI agents for complex decision-making challenges. The algorithm's flexibility and efficiency has led to a lot of essential breakthroughs in changing focus on how AI systems are developed and used for strategic planning and problem solving. Some crucial impacts of MCTS on AI and training AI agents include the capability of making decisions quicker during uncertainty and against a range of opponent strategies, exploring high branching and deep decision trees exhaustively, providing solutions for real-world applications that are unpredictable or have incomplete data such as robotic navigation systems, helps in allocation of resources and most importantly, more research and development has brought more ideas such as the use of hybrid AI models that combines traditional search techniques with machine learning.

3. GAMES

3.1 Tic Tac Toe

Also known as Noughts and Crosses or Xs and Os, Tic Tac Toe is a simple two-player game traditionally played on a 3x3 grid. The game's limited grid size has a relatively small state space making it ideal for developing and testing AI algorithms. Despite its size and simplicity, the fundamentals learned from developing Tic Tac Toe AI are applicable to more complex games; this includes implementing recursive functions and exploring the proficiency of other algorithms in reducing the search space.

Tic Tac Toe has served as a fundamental model in numerous research studies to assess and demonstrate the efficiency of the MCTS algorithm. One notable study has proposed a dynamic sampling tree policy that is designed to optimise the allocation of computational resources within MCTS [1]. The policy focuses on maximising the probability of selecting the best action at the root node by enhancing the overall efficiency of the search process. This approach was validated through experimental evaluations mainly in Tic Tac Toe, demonstrating that the policy surpasses other methods in terms of efficiency. Hence despite Tic Tac Toe's simplicity, it provides a valuable benchmark for developing and assessing advanced MCTS strategies.

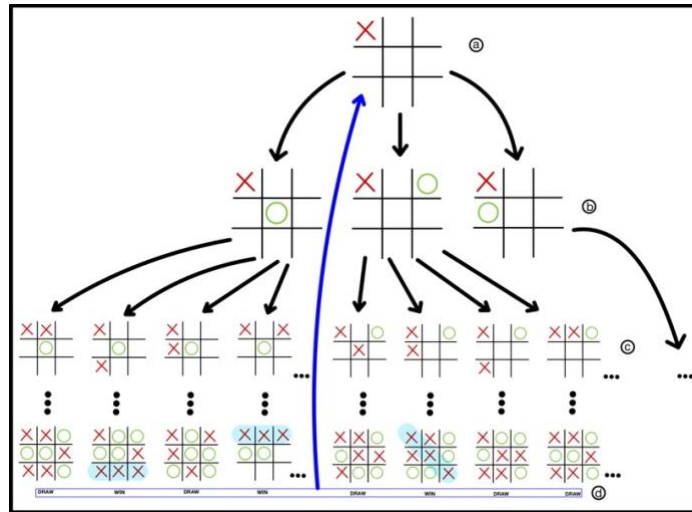


Fig 2: MCTS in Tic Tac Toe gameplay: a) Selection of the most promising node as MCTS makes first move. b) Expansion of child nodes to explore. c) Simulation of random numerous games of the newly expanded nodes, results are recorded. d) Backpropagation of win, lose and draw results that were recorded in Simulation.

3.1.1 The Rules and Strategies of Tic Tac Toe

Traditional Tic Tac Toe is a two-player 3x3 board game where players take turns making a move with either the X or O symbol (each player is provided with only three pieces for the game). Players switch turns making a move on the empty grid spaces and once the move has been made, it cannot be changed until the game has ended. The first player to get their symbols in a row, whether horizontally, vertically or diagonally, wins the game. Lastly, if all the nine grid spaces have been filled and no player has three symbols in a row, the game ends in a draw and a new game may begin. Key strategies include taking

control of the centre, creating two potential winning moves simultaneously in one move and ensuring your opponent does not win by blocking their moves.

3.2 Connect Four

Also known as Four in a Row or Plot 4 and regarded as “the perfect blend of vertical checkers and Tic Tac Toe” [10], Connect Four is a two-player 7x6 grid board game which involves dropping coloured discs into a vertically suspended grid with the aim of connecting their coloured discs vertically, horizontally or diagonally to win. The first-player-win solved conclusion by Victor Allis detects that the first player can force a win on or before the 41st move by starting in the middle column.

Researchers are always looking for a testbed for developing, assessing and comparing MCTS strategies, and as much as games like Tic Tac Toe provide the foundation, they do not provide the vast search space that Connect Four have. With about 4.5 trillion possible different board arrangements in Connect Four [10], it has served as a benchmark for assessing MCTS with other algorithms, for example, the creation of an evolutionary framework that evaluates the performance of Minimax [2], Q-Learning [2] and MCTS algorithms in Connect Four, which, at the end showed that MCTS has the highest win percentage. The game has also been used as a test case to study the theoretical properties of MCTS, contributing to the advancement in AI and game theory research.

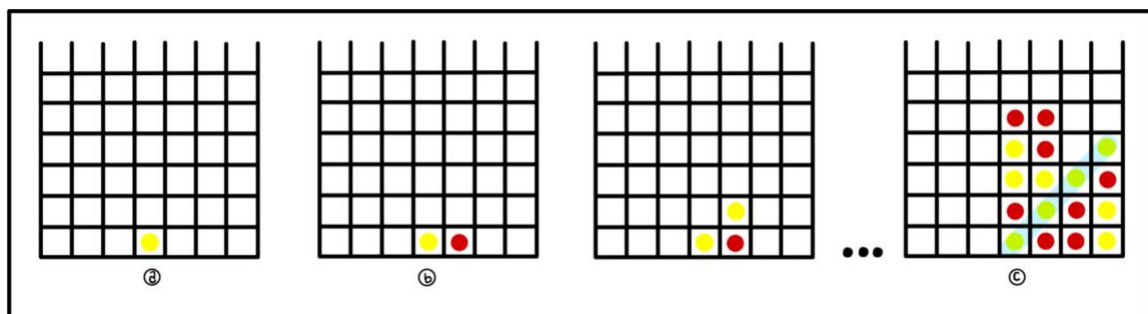


Fig 3: Connect Four Gameplay a) Player 1 makes their first move with the YELLOW disc. b) Player 2 makes their first move with the RED disc. c) After several moves by both players, Player 1 wins the game with four discs in a row diagonally

3.2.1 The Rules and Strategies of Connect Four

The traditional 7x6 Connect Four grid game’s main objective is to have four of the same colour discs in a row, either vertically, horizontally or diagonally. Each player can only drop one disc at a time, the two players alternate turns dropping one of their discs at a time into an unfilled column. If the grid board is filled up before any player gets four of their discs in a row, then the game ends in a draw.

The main strategies include central control which involves controlling the centre of the grid board mainly by making your first drop right in the middle of the board. Other notable strategies are forced moves (creating two potential win scenarios that force your opponent to react), horizontal play (easier to achieve and are less noticeable, and once they are established, they are harder to block from both ends) and block and counter (creating dual three-in-a-row threats that encourage and increase winning chances).

4. IMPLEMENTATION

4.1 Monte-Carlo Tree Search in Tic Tac Toe

Monte-Carlo Tree Search was first used in Tic Tac Toe around the mid-2000s [9], the simplicity and size of the game has made it a widely researched game in computational intelligence for testing and implementing new approaches. The creation of this Tic Tac Toe project mainly consists of three java classes, namely:

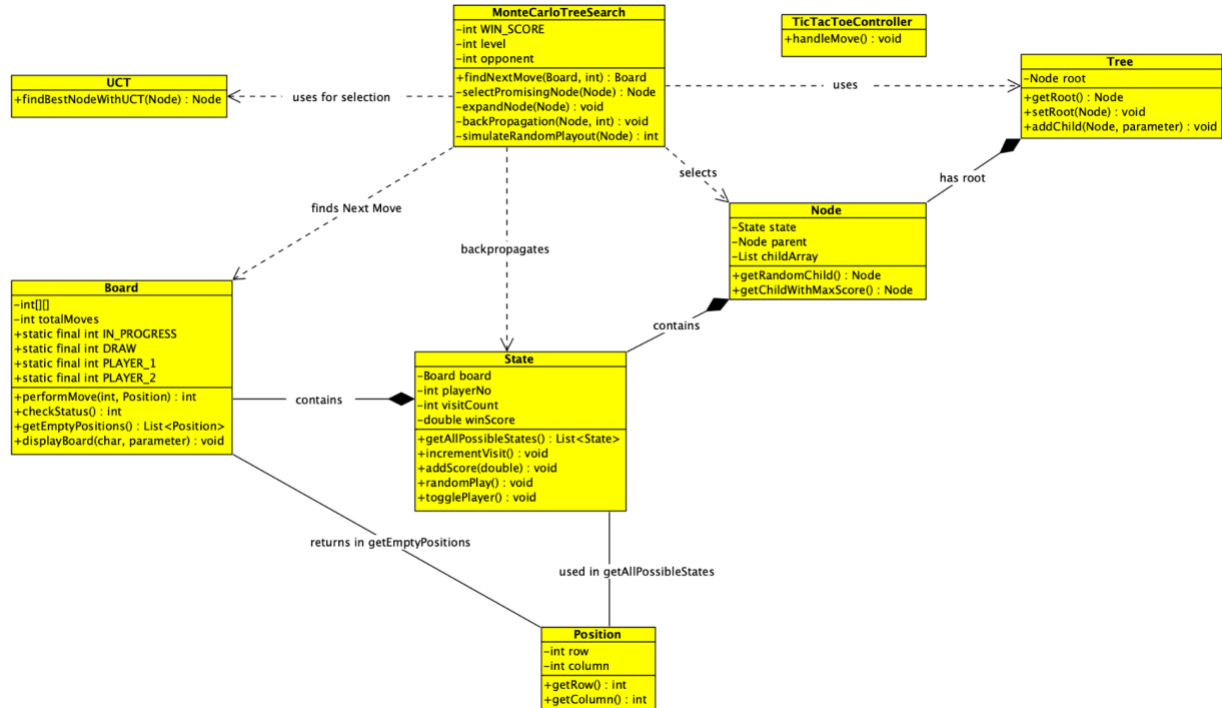


Fig 4: UML Diagram of the Tic Tac Toe backend application: represents all the classes for training the AI agent. These classes provide all the functionality and logic involved with the game.

Board class

The board class plays one of the most integral roles in defining and handling the logic of how the game is played and evaluated, forming the backend framework for building gameplay tactics and AI decisions. It provides foundational structure to manage the game's state such as providing the game pieces (Xs and Os), tracking each player's moves, evaluating the status of the game. The class provides gameplay and utility methods such as:

- *performMove*: makes a game move depending on the player and availability of the grid position on the board and then updates the game status.
- *checkStatus*: check all rows, columns and diagonals to determine if each of the players has won, drawn or the game is still in progress.
- *getEmptyPositions*: returns all empty grid positions on the board to help MCTS determine future possible moves during game state.
- *displayboard*: prints the board to the console during play.

- *printStatus*: prints the output of the game status (win, lose, draw or game in progress).
- *getBoardValues*: getter and setter for the board's state, board can then be retrieved and manipulated.

MCTS class

The implementation of the MCTS algorithm on the game is mainly based on the integration of a four-step process that encourages selection of better winning choices, simulation of many possible game actions and selecting the most promising one. This is achieved by building a tree where each node represents a possible game move and MCTS gets to explore each game state and keep track of the success scores for winning the game.

Beginning at the root node (current game state), the algorithm traverses the tree and selects the best child nodes until it reaches a leaf node (node with no child) using UCB1. UCB1 algorithm provides a better layout score by balancing exploitation of known rewards with the exploration of unvisited nodes. This will help MCTS choose one move to explore based on the success rate and unexplored potential of the nine grid spaces on the game's board. If, for example, we have two blank spaces left in the Tic Tac Toe board (reach a leaf node), one or more child nodes are added for each of these possible moves yet to be explored and the tree search is expanded. Starting from the new child nodes, simulations randomly play the game from the node's board position until a win, lose or draw has been reached (termination state) using the default policy. When simulation reaches termination state, the win/lose result is propagated back up the tree and the scores of all nodes in the path of that node are updated and can later be used to improve accuracy in node selection. After the iteration has reached termination, the algorithm selects the child of the root with the highest win score as the next move to execute in the game.

State class

The state class contains gameplay data such as board layout, player's turn, visit count and the win score that MCTS needs to apply its methods. It helps calculate the best moves by analysing simulated outcomes from numerous possible game states. Notable methods include:

- *getAllPossibleStates*: generates all possible future states from the current state. Key for the expansion phase where new nodes are added to the tree.
- *incrementVisit*: increases the visit count of the current state. Key for the backpropagation phase to update visit count scores after simulation.
- *addScore*: adds win, draw or lose score to the current state's win score. Key for the backpropagation phase to update the score based on simulation outcomes.
- *randomPlay*: carries out random moves on empty grid spaces on the board. Key for the simulation phase for simulating gameplay.

Setting up and configuring the Tic Tac Toe game was a learning curve that had its challenges, the main one being implementing the MCTS algorithm. After my first attempt of implementing and further testing of the MCTS algorithm's optimality and performance, the AI player could not learn from previous scenarios or game states hence the human player could win with ease. Upon several attempts to improve the algorithm by changing constraints such as time and number of simulations, I was unsuccessful, as a result I obtained code on GitHub that I reused which helped optimise the functionality and efficiency of my game.

Other notable challenges encountered during implementation include:

- Setting up Spring Boot for connecting the backend functionality and logic to the frontend presentation and user interface which led to the consideration of using a different framework such as JHipster. However, through more research on the Spring Boot¹ website and Stack Overflow and the use of GenAI copilot I managed to troubleshoot it and set up the framework. Copilot's autocomplete function was instrumental in assisting the setup of my backend controllers which are responsible for facilitating the functionality of the backend to the frontend.
- Logical and gameplay challenges such as ending of gameplay by both players when the board's grid spaces are all occupied.
- OpenCV: part of the project was to create a real-time projectable view for gameplay hence OpenCV was introduced. OpenCV is a computer vision library used for recognising objects for counting and object tracking. Some industries and companies use it for autonomous driving technology to detect road objects and lanes and for analysing medical images to support human diagnosis. In this project the goal was to use it for detecting hand gestures as they pointed to a grid in the game's board to make a move. This would have been done through accessing a webcam and capturing video frames in real-time. The main issue challenge was finding and loading the required libraries as there is less documentation available for MacOS systems. As a result, the alternative of implementing MCTS in Connect Four was taken.
- Creating and implementing classes for Connect Four was made easier by the fact that when using MCTS there is no need to change class names, and they all perform the same functions.

4.2 Monte-Carlo Tree Search in Connect Four

For almost 20 years now, the MCTS algorithm has been used widely in games and AI research. In games like Connect Four the algorithm has the ability to handle the complexities without intensive search or comprehensive domain knowledge. Having been solved mathematically in 1988 by James Allen and Victor Allis [3], it was inevitable that Connect Four would then be trained by an AI algorithm such as MCTS, hence in the late-2000s it happened.

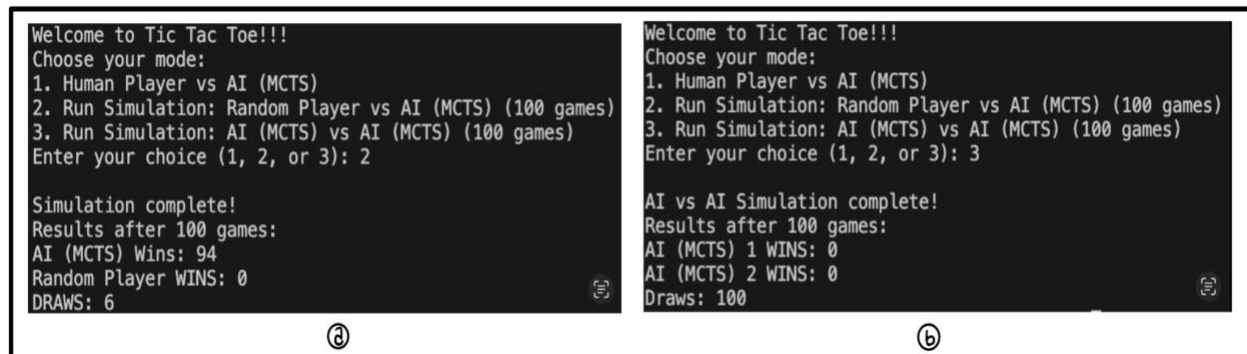
Similar to the Tic Tac Toe game, Connect Four also has three main classes that are fundamental to its logic and gameplay. The board class encapsulates all functionality required in the game. It handles game state updates, move validation, win conditions, and provides utility methods for displaying the game state and determining the game's outcome. The state class encases the state of the game including the board configuration, the current player and the win score and visit count statistics used in the MCTS algorithm to evaluate the success of each move.

Connect Four has differences with Tic Tac Toe such as the grid space size, dropping of discs to the lowest available row and a four-in-a-row condition to win. The most notable difference is how MCTS functions; to ensure more efficiency in finding the next best node during simulation the algorithm blocks an opponent's winning move to prevent instant loss, prioritises creating "connect-three" and "double-three" moves leaving it with only one more move and/or two options to win and most importantly, prioritises the centre column for flexibility (the centre move offers unparalleled flexibility to extend moves across the board effectively).

¹ Spring boot documentation for setting up the backend of the project: <https://docs.spring.io/spring-boot/documentation.html>

5. TESTING AND RESULTS

In order to assess the effectiveness of the MCTS algorithm as an AI strategy for games like Tic Tac Toe and Connect Four, several tests were conducted. These tests evaluate the performance, flexibility and optimality of the AI player under different conditions. Firstly, to test for performance, I ran a simulation of 100 games of AI against a random player where the player to make the first move was randomly selected to determine the consistency of the algorithm and as a result MCTS AI outperformed its opponent. MCTS AI won 94 games and the remaining 6 ended in a draw. However, whenever MCTS made the first move, it won more games. For example, a simulation of 100 games was run and MCTS won 97 games and the remaining 3 ended in a draw proving how effective MCTS is.



```

Welcome to Tic Tac Toe!!!
Choose your mode:
1. Human Player vs AI (MCTS)
2. Run Simulation: Random Player vs AI (MCTS) (100 games)
3. Run Simulation: AI (MCTS) vs AI (MCTS) (100 games)
Enter your choice (1, 2, or 3): 2

Simulation complete!
Results after 100 games:
AI (MCTS) Wins: 94
Random Player WINS: 0
DRAWS: 6

Welcome to Tic Tac Toe!!!
Choose your mode:
1. Human Player vs AI (MCTS)
2. Run Simulation: Random Player vs AI (MCTS) (100 games)
3. Run Simulation: AI (MCTS) vs AI (MCTS) (100 games)
Enter your choice (1, 2, or 3): 3

AI vs AI Simulation complete!
Results after 100 games:
AI (MCTS) 1 WINS: 0
AI (MCTS) 2 WINS: 0
Draws: 100
```

Fig 5: Simulation Test Results a) 100 games of a Random player vs AI were simulated and as a result the AI won 94 games and the remaining 6 ended as a draw displaying the effectiveness of MCTS in Tic Tac Toe. b) A simulation of 100 games where AI played against AI with both AIs consisting of the same number of simulations and UCT constraints was run and as a result this ended in a draw.

Furthermore, an AI vs AI simulation was run to assess the consistency and decision-making of the algorithm and as expected, when two identically fine-tuned AIs play against each other, the games mostly end in a draw, reinforcing the idea that MCTS operates optimally when both agents have the same resources.

To analyse the algorithm's flexibility, human players of different skills in the Tic Tac Toe game were tested against MCTS where the player to make the first move is selected randomly and as a result, beginner players lost most games, intermediate players were able to get a draw but did not manage a win and the expert players managed to force consistent draws. These findings indicate that MCTS can compete at an intermediate-to-expert level but may still find it challenging against highly optimised human strategies.

Lastly, to test for optimality, a number of tests were conducted by adjusting the number of simulations and exploration constants in MCTS and the results showed higher simulations (1000+) led to stronger gameplay but needed more resources such as processing time, lower exploration constants (0.6) led to a greedy AI, causing it to miss long-term strategies and only look for quick ways to win and higher exploration constants (3.0) made AI more random, reducing its winning rate, hence proving that modifying UCT parameters is vital for maximising AI performance.

6. CONCLUSION

Generations of algorithms have aimed to master complex decision-making tasks especially in board games such as Tic Tac Toe and Connect Four. One of the most distinct due to its performance and adaptability across different search spaces is the Monte-Carlo Tree Search algorithm. Traditional algorithms, notably, Minimax, mostly rely on clear and accurate heuristics, limiting their effectiveness and adaptability. On the other hand, MCTS uses simulation to dynamically assess game states, enabling greater performance without a significant dependence on manually created rules.

Experimental findings demonstrate the efficiency of MCTS, which also shows that it can dynamically modify strategies based on the complexity of the game's state space. MCTS consistently demonstrates greater performance against random players and offers human opponents a challenging game. The versatility of MCTS allows for significant improvements through increased computational resources, integration with reinforcement learning, neural network evaluations or using hybrid approaches based on popular models like AlphaZero.

Overall, this study highlights the effectiveness and adaptability of MCTS, demonstrating the potential to serve as a fundamental approach for a wide range of AI applications that go well beyond conventional board games and into numerous complex decision-making. Understanding how to achieve high performance with MCTS in games like Connect Four and Tic Tac Toe provides new possibilities for high-performance AI in a wide range of challenging problems.

REFERENCES

- [1] Zhang, G., Peng, Y. & Xu, Y. (2022). An Efficient Dynamic Sampling Policy for Monte Carlo Tree Search. [Manuscript submitted for publication]. Department of Management Science and Information Systems, Peking University & Department of Computer Science, Beijing Jiaotong University.
- [2] Taylor, H. & Stella, L. (2024). *An Evolutionary Framework for Connect-4 as Test-Bed for Comparison of Advanced Minimax, Q-Learning and MCTS*.
- [3] Allen, J. and Allis, V. (1988) *A Knowledge-based Approach of Connect-Four: The Game is Solved*. Master's Thesis, Vrije Universiteit Amsterdam.
- [4] B. Bouzy, B. Helmstetter, Monte-Carlo Go developments, in: 10th Advances in Computer Games Conference, pp. 159-174.
- [5] Gelly, S. & Silver, D. (2011). Monte-Carlo tree search and rapid action value estimation in computer Go. *Artificial Intelligence*, 175.
- [6] Kocsis, L. & Szepesvári, C. (2006). Bandit based Monte-Carlo planning. *Machine Learning: ECML 2006*, Berlin, pp. 282-293.
- [7] Czech, J., Korus, P. & Kersting, K. (2020). Monte-Carlo Graph Search for AlphaZero. Technical University of Darmstadt, Germany, pp 2-3
- [8] Vogt, M.D.P.-T. (2020). Applying Monte Carlo Search and Monte Carlo Tree Search on Embedded Systems to Play Connect Four with a Robotic Arm. Bachelor Thesis. University of Applied Sciences Mannheim.
- [9] Chaslot, G., Bakkes, S., Szita, I. & Spronck, P. (2008). Monte-Carlo Tree Search: A New Framework for Game AI. *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference*, pp. 216-217.
- [10] Elakai Outdoor. (n.d.). *The History of Connect 4: From Its Inception to Becoming a Classic*. Available at: https://elakaioutdoor.com/blogs/lifestyle/the-history-of-connect-4-from-its-inception-to-becoming-a-classic?srltid=AfmBOorF37AwFcCfj8q18rpIUQrDGvTbYxQXw_1z3H3uGKgaB07w36qg