# INFO 6205 - Program Structures and Algorithms Spring 2024

## Team Members:

NUID: 002209157 - Khushank Mistry
NUID: 002871241 - Pranav Arun Kapse
NUID: 002273113 - Amar Nagargoje

GitHub Link: https://github.com/amarneu/INFO6205-MCTS-FinalProject

**TASK:** Write a program to implement a Monte Carlo Tree Search for two games:

• Tic-tac-toe.

• Connect4

## Aim:

The aim of this project report is to implement and analyze the Monte Carlo Tree Search (MCTS) algorithm for two classic games: Tic-tac-toe and Connect Four. Through this report, we aim to provide a comprehensive understanding of how MCTS works in the context of these games, along with the implementation details, optimizations, and performance evaluations.

MCTS is a powerful technique for game-playing AI, which uses randomized simulations to search the game tree efficiently. The report details the implementation of MCTS for both games, including the structure of the algorithm, optimizations applied, and performance evaluations.

# Approach:

1. **Game Representation**: Represent the Connect Four game board as a 2D array or matrix, where each cell can hold the value of the player's token or an empty cell.
2. **Node Structure**: Define a node structure to represent the game state in the MCTS tree. Each node should contain the current game board configuration, the active player, and other relevant information.
3. **Selection**: Implement the selection phase of the MCTS algorithm, which involves traversing the tree from the root node to a leaf node. Use the Upper Confidence Bound (UCB) formula or a similar heuristic to balance exploration and exploitation when selecting the child nodes.
4. **Expansion**: At the leaf node, generate all possible legal moves for the current player. Create new child nodes for each legal move, representing the resulting game board configurations.
5. **Simulation**: From the newly expanded nodes, perform a simulation (also known as rollout or playout) by making random legal moves until a terminal state (win, loss, or draw) is reached. This simulation phase helps estimate the potential outcome of each move.
6. **Backpropagation**: After the simulation, update the statistics (e.g., visit count, win count) of the nodes along the path from the leaf node to the root node based on the simulation result. This backpropagation phase reinforces the promising moves and guides the search towards better decisions.
7. **Iteration**: Repeat the selection, expansion, simulation, and backpropagation phases for a predetermined number of iterations or until a computational budget is exhausted.
8. **Move Selection**: After the iterations, select the most promising move from the root node based on the accumulated statistics (e.g., the child node with the highest win rate or visit count).
9. **Game Loop**: Implement a game loop where the MCTS algorithm is executed for each player's turn. Update the game board with the selected move, and check for win, loss, or draw conditions.
10. **Parallelization (Optional)**: Explore parallelization techniques, such as multi-threading or distributed computing, to speed up the MCTS simulations and improve the overall performance.
11. **Enhancements (Optional)**: Investigate and implement additional enhancements to the MCTS algorithm, such as progressive bias, rapid action value estimation (RAVE), or other domain-specific heuristics to improve the algorithm's efficiency and decision-making capabilities.
12. **Testing and Evaluation**: Conduct thorough testing of the MCTS implementation, including unit tests and performance benchmarking. Evaluate the algorithm's effectiveness by playing against other AI players or human opponents.

# Invariants

1. **Board Dimensions**:
   - The number of rows (numRows) and columns (numColumns) in the game board should always be positive integers.
   - The dimensions of the gameboard 2D array should match the numRows and numColumns values.
2. **Board Initialization**:
   - After initializing the game board, all cells should be set to the '.' character, representing an empty cell.
3. **Valid Token Characters**:
   - The token characters used by players should be single, non-whitespace characters.
   - The token characters for the two players should be different.
4. **Valid Column Index**:
   - The column index passed to the placeToken method should be within the valid range (0 to numColumns - 1).
5. **Token Placement**:
   - Tokens can only be placed in empty cells (cells with the '.' character).
   - Tokens should be placed in the lowest available row for the chosen column.
6. **Game State**:
   - The game should have exactly two players at any given time.
   - Only one player should be designated as the active player at a time.
   - The game should not be over (no winner or draw) until all cells are filled or a winning condition is met.
7. **Win Condition**:
   - A player wins if they have four consecutive tokens (horizontally, vertically, or diagonally) on the board.
   - There can be at most one winner in a game.
8. **Draw Condition**:
   - The game should end in a draw if all cells on the board are filled, and no player has achieved a winning condition.
9. **Player Equality**:
   - Two players should be considered equal if and only if they have the same token character.
10. **Player Uniqueness**:
    - The hashCode method for players should return unique values for players with different token characters.

These invariants represent the expected properties and constraints that should hold true throughout the execution of the Connect Four game. Verifying and maintaining these invariants can help ensure the correctness and robustness of the implementation.
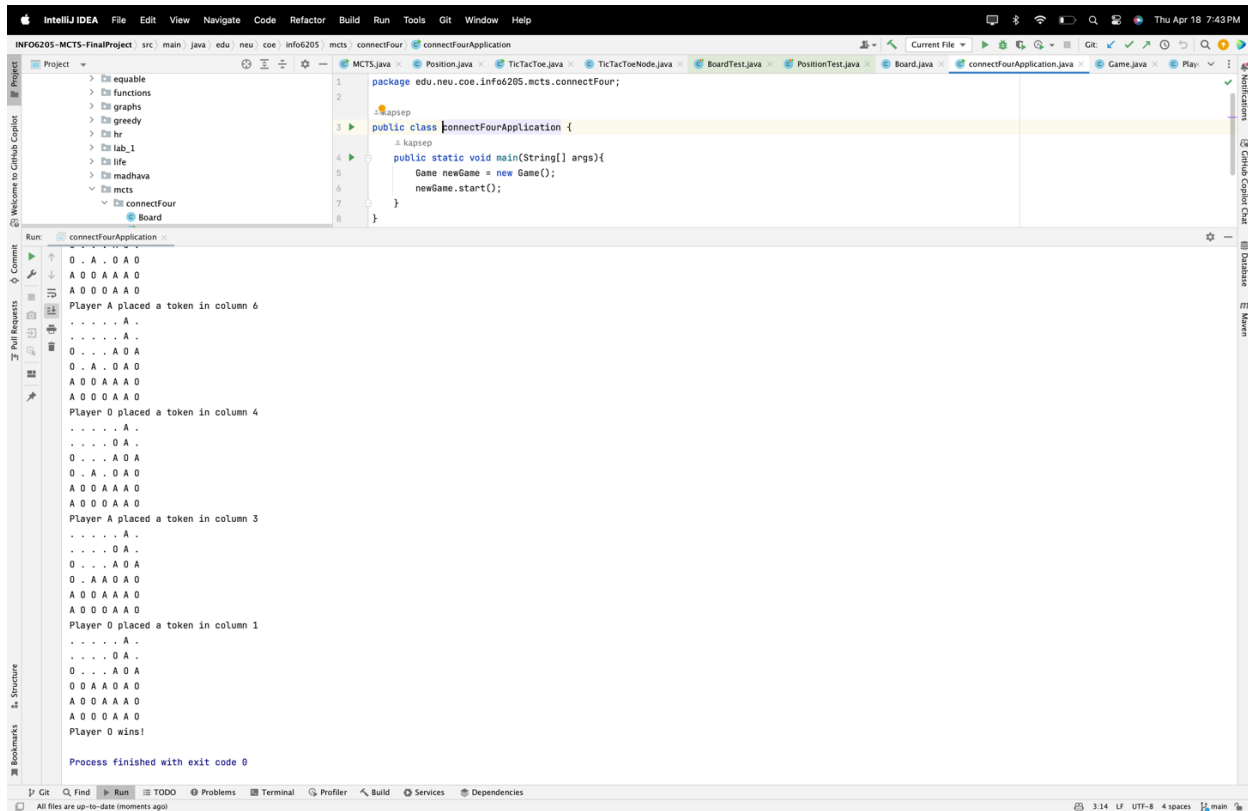
# Connect4 - Code Snippet

The provided code implements a Connect Four game using the Monte Carlo Tree Search (MCTS) algorithm. The game is played between two AI players, and the MCTS algorithm is used to explore the game tree and make optimal moves. The code includes classes for the game board, players, and the game itself, as well as unit tests for each class.

The Board class represents the game board and provides methods for placing tokens, checking for wins, displaying the board, and managing the game board state. The Player class represents a player with a character token and includes methods for comparing players based on their tokens.

The Game class manages the game state, including the board, players, and active player. It handles the game loop, making moves, checking for wins or draws, and switching the active player.

The unit tests cover various scenarios for the Board, Game, and Player classes, ensuring the correctness of their methods and behavior.

## Game Output

- **Board**

```java
public boolean checkVictory(char token) {
    // Check if the specified token has achieved a winning condition
    // Check for horizontal wins
    for (int row = 0; row < numRows; row++) {
        for (int col = 0; col < numColumns - 3; col++) {
            if (gameBoard[row][col] == token && gameBoard[row][col + 1] == token &&
                    gameBoard[row][col + 2] == token && gameBoard[row][col + 3] == token) {
                return true; // Horizontal win detected
            }
        }
    }

    // Check for vertical wins
    for (int col = 0; col < numColumns; col++) {
        for (int row = 0; row < numRows - 3; row++) {
            if (gameBoard[row][col] == token && gameBoard[row + 1][col] == token &&
                    gameBoard[row + 2][col] == token && gameBoard[row + 3][col] == token) {
                return true; // Vertical win detected
            }
        }
    }

    // Check for positive diagonal wins
    for (int row = 3; row < numRows; row++) {
        for (int col = 0; col < numColumns - 3; col++) {
            if (gameBoard[row][col] == token && gameBoard[row - 1][col + 1] == token &&
                    gameBoard[row - 2][col + 2] == token && gameBoard[row - 3][col + 3] == token) {
                return true; // Positive diagonal win detected
            }
        }
    }

    // Check for negative diagonal wins
    for (int row = 0; row < numRows - 3; row++) {
        for (int col = 0; col < numColumns - 3; col++) {
            if (gameBoard[row][col] == token && gameBoard[row + 1][col + 1] == token &&
                    gameBoard[row + 2][col + 2] == token && gameBoard[row + 3][col + 3] == token) {
                return true; // Negative diagonal win detected
            }
        }
    }

    return false; // No winning condition found
}
```

The checkVictory method checks if the specified token has achieved a winning condition on the game board by iterating through the board and checking for four consecutive tokens horizontally, vertically, and diagonally (positive and negative slopes). If a winning condition is found, the method returns true; otherwise, it returns false.

```java
public boolean placeToken(int column, char token) {
    // Place a token in the specified column
    if (column < 0 || column >= numColumns) {
        // Invalid column, return false
        return false;
    }
    for (int row = numRows - 1; row >= 0; row--) {
        if (gameBoard[row][column] == '.') {
            gameBoard[row][column] = token;
            return true; // Token placed successfully
        }
    }
    return false; // Column is full, token cannot be placed
}
```

The placeToken method attempts to place a token in the specified column of the game board. It first checks if the column is valid, and if so, it iterates from the bottom row upwards to find the first available cell. If a cell is found, the token is placed there, and the method returns true. If the column is full or invalid, the method returns false.

- Game

```java
boolean takeTurn() {
    // Take a turn in the game
    board.displayBoard(); // Display the current board state
    int columnChoice = rng.nextInt( bound: 7); // Randomly choose a column (assuming 7 columns)
    if (!board.placeToken(columnChoice, activePlayer.getToken())) {
        // If the chosen column is full, try again
        System.out.println("Column " + columnChoice + " is full, try again.");
        return false;
    }

    System.out.println("Player " + activePlayer.getToken() + " placed a token in column " + columnChoice);

    if (board.checkVictory(activePlayer.getToken())) {
        // If the active player has won, display the board and announce the winner
        board.displayBoard();
        System.out.println("Player " + activePlayer.getToken() + " wins!");
        return true;
    } else if (board.isGridFull()) {
        // If the board is full and no one has won, it's a draw
        board.displayBoard();
        System.out.println("The game is a draw!");
        return true;
    }

    // Switch to the next player
    switchActivePlayer();
    return false;
}
```

The takeTurn method handles a single turn in the game. It displays the current board state, randomly selects a column to place the active player's token, checks for a win or draw condition, and switches to the next player if the game is not over. The method returns true if the game is over (win or draw), and false otherwise.
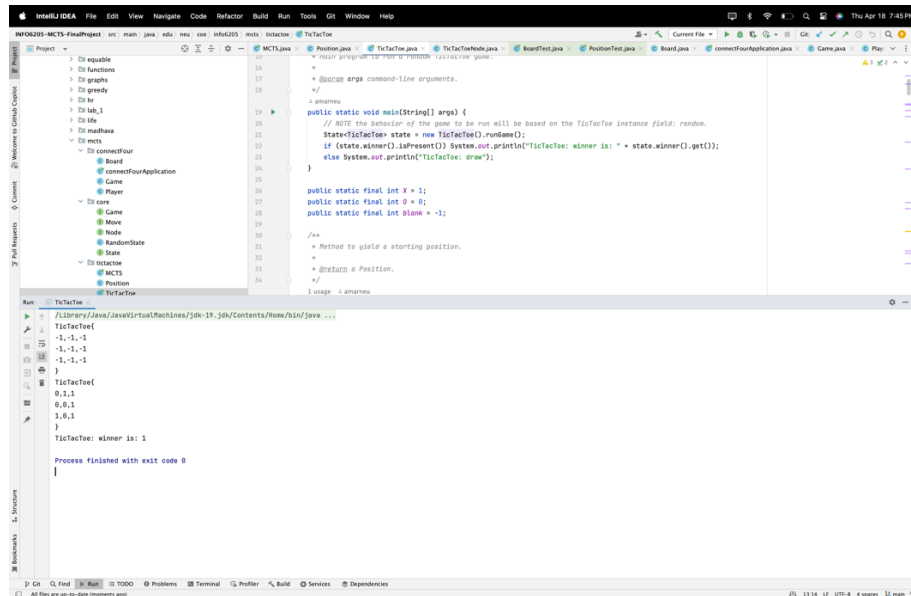
## Tic Tac Toe - Code Snippet

The code implements a Tic Tac Toe game where two AI players compete against each other using the MCTS algorithm. The MCTS algorithm is used to explore the game tree and make optimal moves. The implementation includes the following key components:

1. TicTacToe class: Represents the game state and provides methods for making moves and checking for a winner.
2. TicTacToeNode class: Extends the Node class and represents a node in the game tree.
3. MCTS class: Implements the Monte Carlo Tree Search algorithm, including methods for selection, simulation, backpropagation, and determining the optimal move.

The MCTS class contains the main method, which sets up the game, creates the root node, and performs a specified number of simulations. During each simulation, the algorithm selects the optimal child node, simulates the game from that node, backpropagates the result, and determines the winning node and player.

- ## MCTS (Monte Carlo Tree Search)

## Game Output



```java
public static void main(String[] args) {

    // Machine vs Machine

    TicTacToe game = new TicTacToe();
    root = new TicTacToeNode(game.start());
    MCTS mcts = new MCTS(root);

    for (int i = 0; i < SIMULATION_LIMIT; i++) {
        // Selection
        Node<TicTacToe> node = mcts.selectOptimalChildNode(root);
        // Simulation
        int result = mcts.simulate(node);
        // Backpropagation
        mcts.backpropagate(node, result);
        // Determine the winning node
        if (root.children().isEmpty()) {
            System.out.println("No moves available.");
        } else {
            Node<TicTacToe> bestMove = mcts.selectOptimalChildNode(root);
            if (bestMove != null) {
                System.out.println("Recommended move: " + bestMove.state().toString());
            } else {
                System.out.println("No best move could be determined.");
            }
        }
        Node<TicTacToe> winningNode = mcts.determineWinningNode(root);
        int winningPlayer = mcts.determineWinner(winningNode);
        System.out.println("Winner: " + (winningPlayer == -1 ? "Draw" : "Player " + winningPlayer));

    }
}
```

Sets up a game of Tic Tac Toe between two AI players using the Monte Carlo Tree Search (MCTS) algorithm. It performs a specified number of simulations, selects the optimal child node (move) at each step, simulates the game from that node, backpropagates the result, and determines the winning node and player at the end of the simulations.

```java
public void backpropagate(Node<TicTacToe> node, int result) {
    while (node != null) {
        int playout = node.playouts();
        node.setPlayouts(playout + 1);
        int wins = node.wins();
        if ((node.state().player() == 1 && result == 1) ||
                (node.state().player() == 0 && result == -1)) {
            node.setWins(wins + 2);
        } else if (result == 0) {
            node.setWins(wins + 1);
        }
        node = node.getParent();
    }
}
```

The method is part of the Monte Carlo Tree Search (MCTS) algorithm. It updates the statistics (number of playouts and wins) of the nodes along the path from the given node to the root node, based on the result of the simulation. This process is known as backpropagation and is crucial for guiding the search towards more promising moves.

```java
public Node<TicTacToe> selectOptimalChildNode(Node<TicTacToe> node) {
    while (!node.isLeaf()) {
        if (!node.children().isEmpty()) {
            node = node.children().stream().max((n1, n2) -> {
                double ucb1 = UCTFormula(n1);
                double ucb2 = UCTFormula(n2);
                return Double.compare(ucb1, ucb2);
            }).get();
        } else {
            // If the node is not a leaf node, then explore the node
            node.explore();
            return node;
        }
    }
    return node;
}
```

The selectOptimalChildNode It recursively selects the child node with the highest Upper Confidence Bound (UCB) value until a leaf node is reached. This selection process is known as the tree policy and aims to balance exploration and exploitation to find the most promising moves.

```java
    double UCTFormula(Node<TicTacToe> node) {
        if (node.playouts() == 0)
            return Double.MAX_VALUE;
        return (double) node.wins() / node.playouts()
                + EXPLORATION_PARAM * Math.sqrt(Math.log(node.playouts()) / node.playouts());
    }
}
```
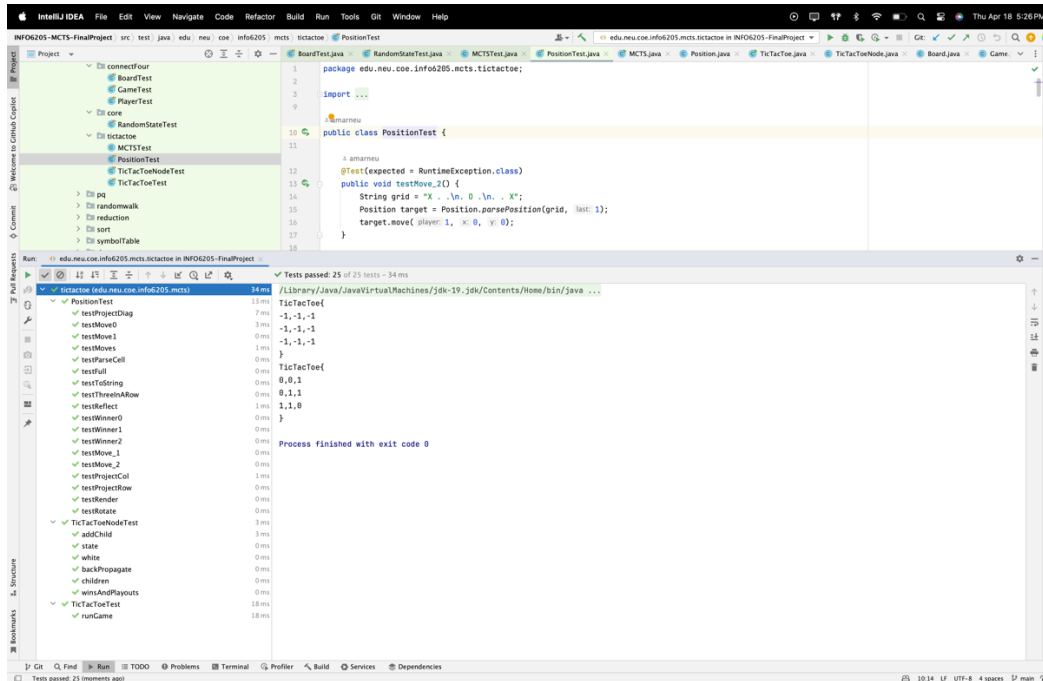
The UCTFormula method calculates the Upper Confidence Bound (UCB) value for a given node in the Monte Carlo Tree Search algorithm, which is used to balance exploration and exploitation during the tree policy.

Test Cases:

- Connect4
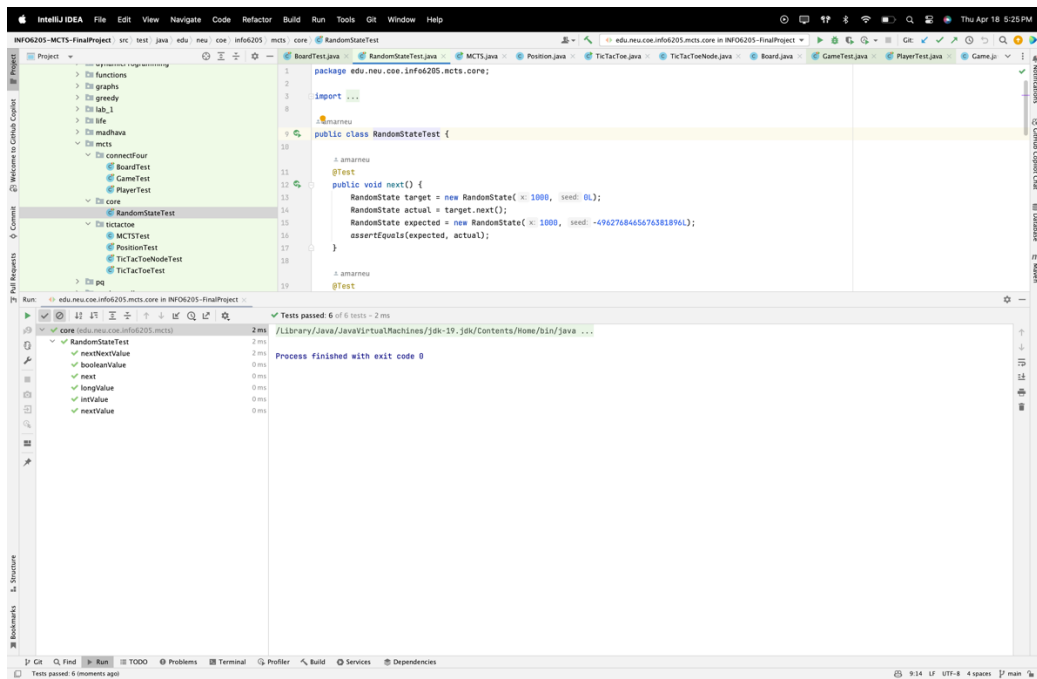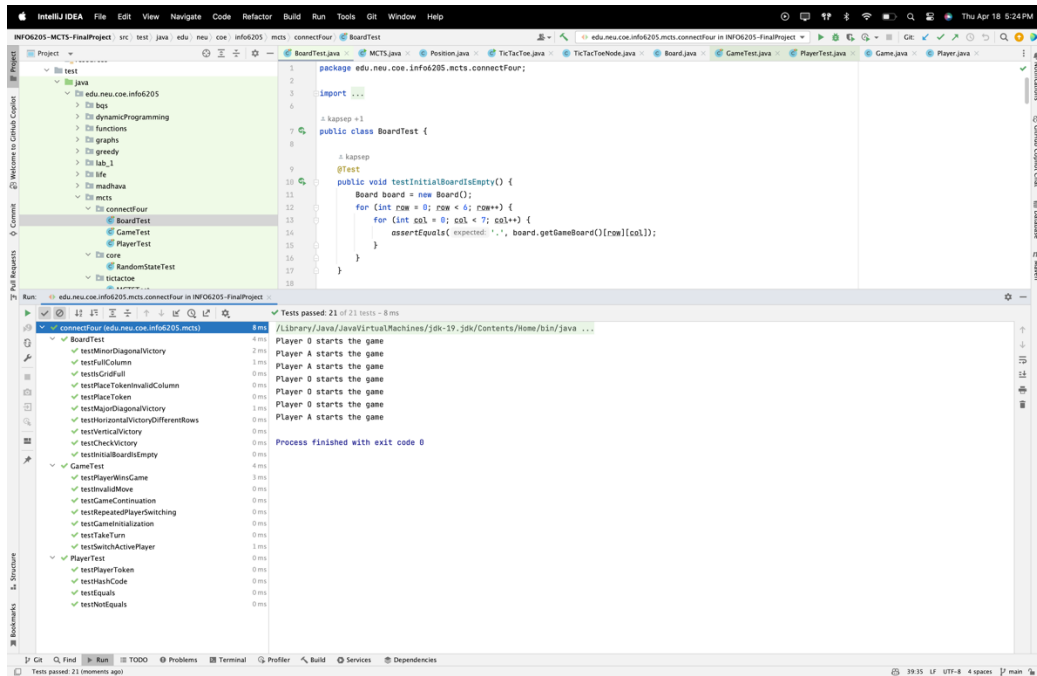
- Tic Tac Toe



# Conclusion

The MCTS algorithm proves to be an effective approach for exploring the game tree and making optimal moves in the Connect Four and Tic Tac Toe games. By combining the game logic with the MCTS algorithm, the implementation enables two AI players to compete against each other, simulating numerous game scenarios and evaluating the most promising moves.

The algorithm's ability to balance exploration and exploitation through the Upper Confidence Bound (UCB) formula allows for efficient traversal of the game tree, leading to intelligent decision-making.

Overall, the MCTS algorithm demonstrates its strength in solving complex problems with large state spaces, making it a valuable tool for developing AI players in games like Connect Four and beyond.

## References

https://web.mit.edu/sp.268/www/2010/connectFourSlides.pdf

https://www.washingtonpost.com/news/wonk/wp/2015/05/08/how-to-win-any-popular-game-according-to-data-scientists/

https://www.harrycodes.com/blog/monte-carlo-tree-search

https://pranav-agarwal-2109.medium.com/game-ai-learning-to-play-connect-4-using-monte-carlo-tree-search-f083d7da451e