Tic-Tac-Toe & Connect4

Program Structure & Algorithms

Team Members:Basavaraj Patil, Saurabh Srivastava, Stafny Velitia Karkada

Github Repo Link:https://github.com/ssaurabh760/INFO6205-FinalProject

Abstract: **This report details the development of a Connect Four game utilizing the Monte Carlo Tree Search (MCTS) algorithm as the foundation for its decision-making process. Initially, our exploration into MCTS was guided by its application in a simpler context through the game of Tic-tac-toe, which served as a preliminary model for understanding the dynamics and implementation strategies of MCTS. This foundational study was instrumental in shaping our approach to developing the Connect Four game. Throughout the project, we encountered several challenges, particularly in adapting MCTS to handle the increased complexity and strategic depth of Connect Four compared to Tic-tac-toe. Additionally, we conducted a comparative analysis between MCTS and a Random search algorithm, which highlighted significant enhancements in AI performance, showcasing MCTS's ability to create a more challenging and engaging game environment. This report not only discusses these challenges and the solutions we implemented but also reflects on the broader implications of using Monte Carlo methods in game AI development, emphasizing the adaptability and scalability of MCTS for enhancing game intelligence and player interaction.**

I.Problem Description

This project aims to gain a deeper understanding of the Monte Carlo Tree Search Algorithm. To demonstrate a working code of Monte Carlo Test Search for a game, and justify the benefits of using the algorithm in our game.

The core objective is to not only implement the MCTS algorithm but also to gain a deep understanding of its mechanisms and why it is effective. This involves dissecting how MCTS balances exploration of new, potentially rewarding moves with the exploitation of known, advantageous strategies.

Below are the key objectives that we would focus on:

Tic-tac-toe: Apply MCTS to a well-understood and relatively simple game to establish a baseline understanding and functionality of the algorithm.

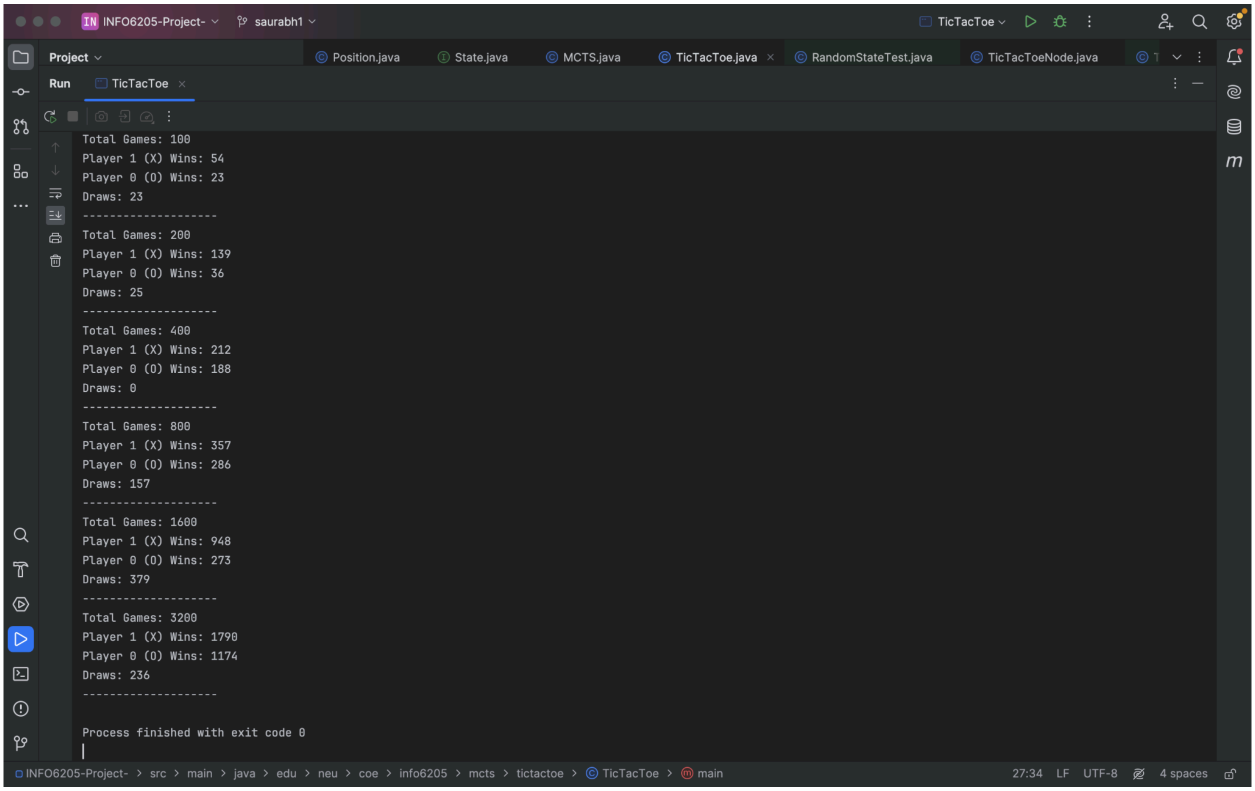
ConnectFour: Extend the application of MCTS to a more complex game, which presents a larger state space and requires more strategic depth, to test the robustness and adaptability of the algorithm.

Performance Analysis: Evaluate the performance of the MCTS algorithm in both games through systematic testing and analysis. This includes assessing the algorithm’s efficiency, accuracy, and scalability.

II.Analysis

The primary objective of using Tic-Tac-Toe was to gain insights into the integration of MCTS within a relatively simpler game environment. This approach allowed us to address and implement the "TO BE IMPLEMENTED" sections in the skeleton code provided. Our implementation involves the Position class, which manages the game's state efficiently. The **moves()** method dynamically generates potential moves for the current player, ensuring gameplay progresses smoothly without manual input for possible moves. The **move()** method applies a player's action to the game board, updating the state based on the chosen move. Additionally, the **threeInARow()** method is designed to check for any winning conditions across rows, columns, and diagonals, encapsulating the game's win logic. We get the below output that shows the tic-tac-toe game running.

We could successfully run the test cases and below are the screenshots for the same.

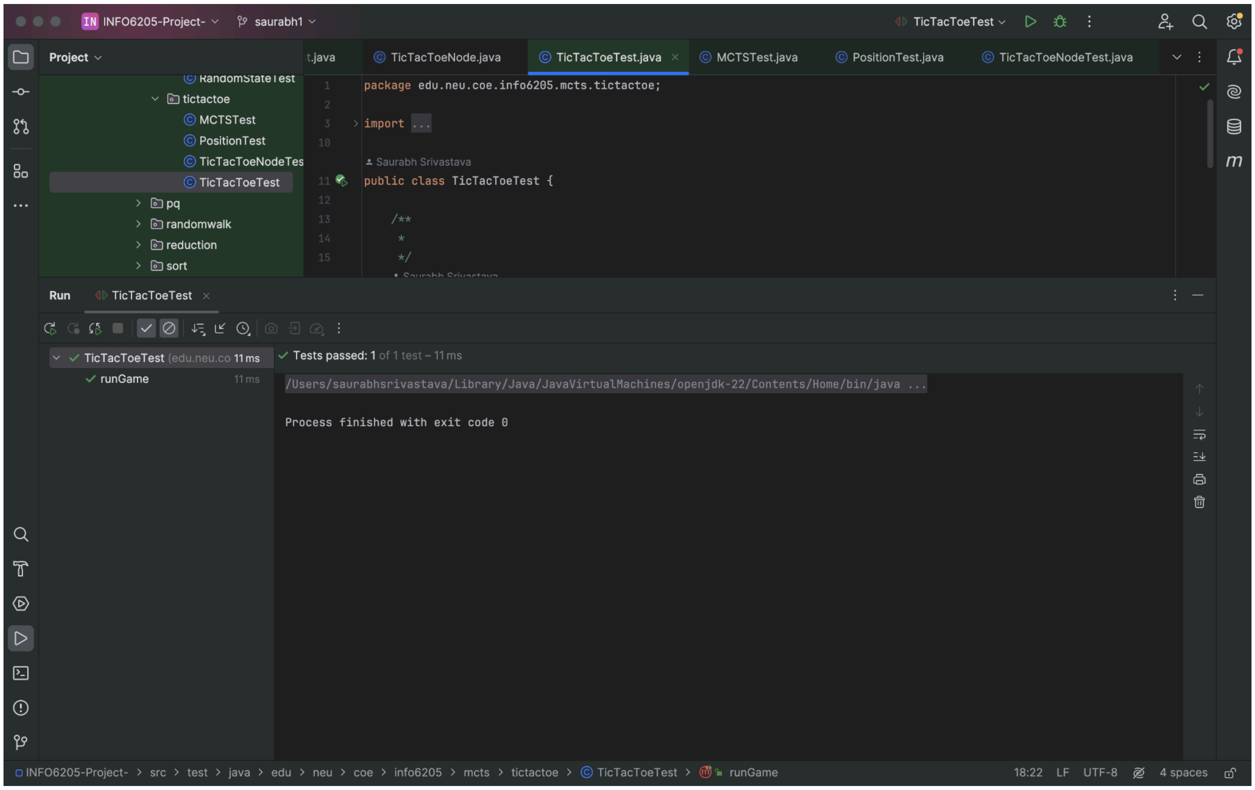


```
Total Games: 100
Player 1 (X) Wins: 54
Player 0 (O) Wins: 23
Draws: 23
--------------------
Total Games: 200
Player 1 (X) Wins: 139
Player 0 (O) Wins: 36
Draws: 25
--------------------
Total Games: 400
Player 1 (X) Wins: 212
Player 0 (O) Wins: 188
Draws: 0
--------------------
Total Games: 800
Player 1 (X) Wins: 357
Player 0 (O) Wins: 286
Draws: 157
--------------------
Total Games: 1600
Player 1 (X) Wins: 948
Player 0 (O) Wins: 273
Draws: 379
--------------------
Total Games: 3200
Player 1 (X) Wins: 1790
Player 0 (O) Wins: 1174
Draws: 236
--------------------

Process finished with exit code 0
```

<u>Running Unit Test cases</u>

We ran the unit test case in the TicTacToeTest file(runGame unit test case), the PositionTest file and TicTacToeNodeTest)

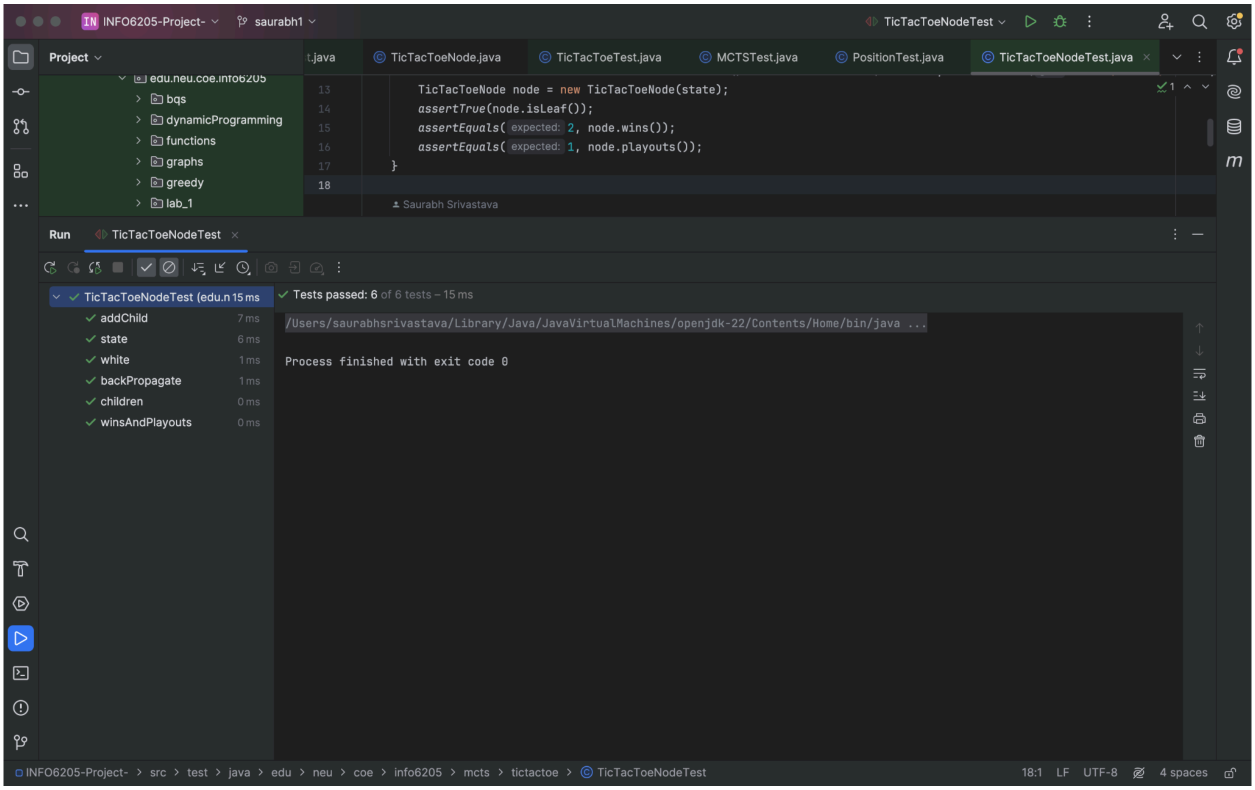
Below are the screenshots of the successful passing of all the test cases



```
package edu.neu.coe.info6205.mcts.tictactoe;

import ...

public class TicTacToeTest {

    /**
     *
     */
```

✓ Tests passed: 1 of 1 test – 11 ms

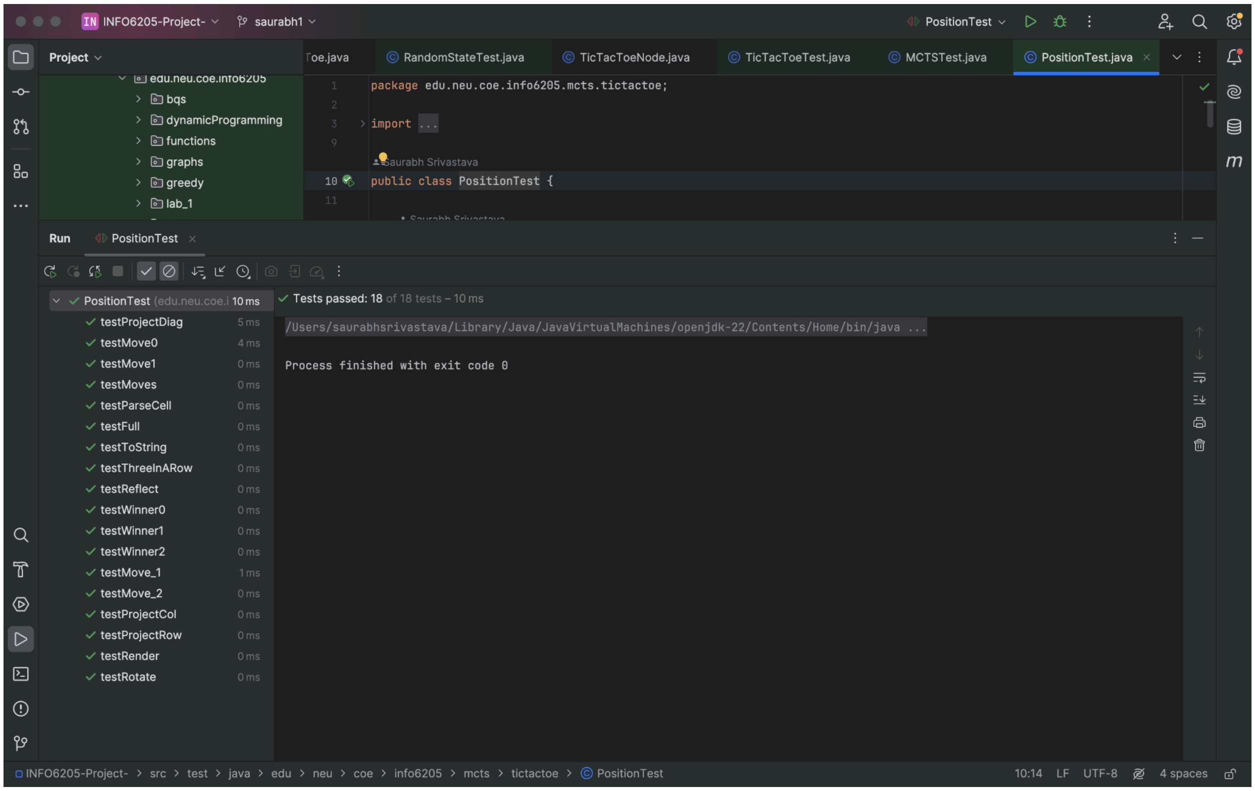
```
/Users/saurabhsrivastava/Library/Java/JavaVirtualMachines/openjdk-22/Contents/Home/bin/java ...

Process finished with exit code 0
```



```
        TicTacToeNode node = new TicTacToeNode(state);
        assertTrue(node.isLeaf());
        assertEquals( expected: 2, node.wins());
        assertEquals( expected: 1, node.playouts());
    }
```

```
✓ TicTacToeNodeTest (edu.n 15 ms
    ✓ addChild          7 ms
    ✓ state             6 ms
    ✓ white             1 ms
    ✓ backPropagate     1 ms
    ✓ children          0 ms
    ✓ winsAndPlayouts   0 ms
```

```
✓ Tests passed: 6 of 6 tests – 15 ms

/Users/saurabhsrivastava/Library/Java/JavaVirtualMachines/openjdk-22/Contents/Home/bin/java ...

Process finished with exit code 0
```



```
package edu.neu.coe.info6205.mcts.tictactoe;

import ...

public class PositionTest {
```

```
✓ PositionTest (edu.neu.coe.i 10 ms
    ✓ testProjectDiag   5 ms
    ✓ testMove0         4 ms
    ✓ testMove1         0 ms
    ✓ testMoves         0 ms
    ✓ testParseCell     0 ms
    ✓ testFull          0 ms
    ✓ testToString      0 ms
    ✓ testThreeInARow   0 ms
    ✓ testReflect       0 ms
    ✓ testWinner0       0 ms
    ✓ testWinner1       0 ms
    ✓ testWinner2       0 ms
    ✓ testMove_1        1 ms
    ✓ testMove_2        0 ms
    ✓ testProjectCol    0 ms
    ✓ testProjectRow    0 ms
    ✓ testRender        0 ms
    ✓ testRotate        0 ms
```

```
✓ Tests passed: 18 of 18 tests – 10 ms

/Users/saurabhsrivastava/Library/Java/JavaVirtualMachines/openjdk-22/Contents/Home/bin/java ...

Process finished with exit code 0
```

Implementing the MCTS Code:

Monte Carlo Tree Search is an algorithm used for making optimal decisions in a given domain by taking random samples in the decision space and building a tree model based on the outcomes. It consists of four main steps:

–**Selection**: Start from the root node and select successive child nodes down to a leaf node using a specific policy, often the Upper Confidence Bound (UCB) applied to trees, which balances exploration and exploitation.
–**Expansion**: Unless the leaf node ends the game decisively (win/loss), create one or more child nodes and choose one to explore.
–**Simulation**: Play a random playout from the new node to a terminal state.
–**Backpropagation**: Use the result of the playout to update information in the nodes on the path from the leaf back up to the root.
The process is iterated upon until a time limit or computation threshold is reached.

Summary of the Material Links:

The provided references, such as the Wikipedia article on MCTS and the review paper from Arxiv, highlight the generic nature of MCTS and its application to a variety of games beyond Tic-tac-toe. They outline the fundamental components of the algorithm, its theoretical underpinnings, and examples of its successful application in domains like board games, video games, and even real-world decision-making scenarios.

The review paper particularly emphasizes the advancements and variations in MCTS, including enhancements that optimize its performance and extend its applicability. The guide offers practical insights into implementing MCTS, demonstrating its broad utility and effectiveness in both academic and commercial applications.

The primary function, **run(int iterations)**, orchestrates the execution of the MCTS algorithm over a defined number of iterations. This function is instrumental in progressively refining the decision-making process of the AI by simulating numerous game scenarios. Each iteration comprises a sequence of steps starting from node selection, game state simulation, and backpropagation of the results.

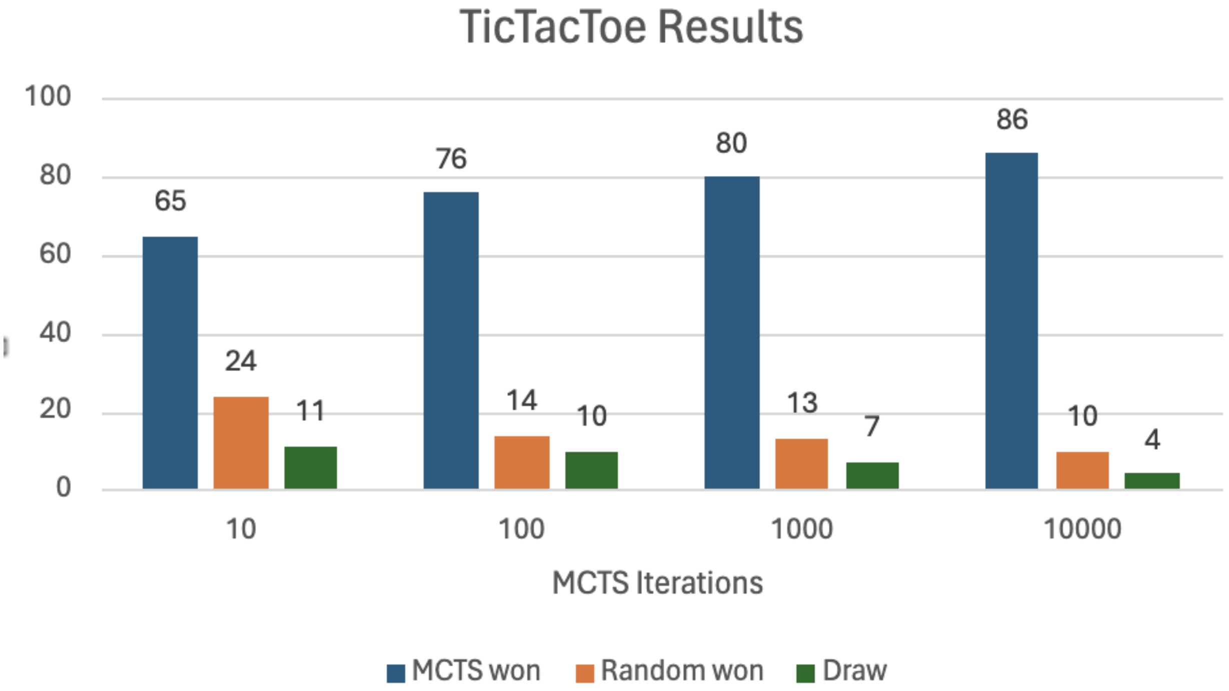
The select(Node<TicTacToe>) method is a critical component of the MCTS architecture. It employs the Upper Confidence Bound applied to Trees (UCT) strategy to select nodes for further exploration. The UCT formula is integral here, striking a balance between exploring under-explored nodes and exploiting nodes with historically favorable outcomes. This dual approach ensures that the search space is navigated efficiently, optimizing both the depth and breadth of the tree exploration.

If all child nodes of the current node are fully expanded, select() method continues to traverse down the tree using the UCT values to guide its path towards the most promising node. Conversely, if there are any nodes yet to be fully explored, it prioritizes these, allowing the algorithm to discover new potential strategies or responses within the game.

Tic-tac-toe could be solved by other simple algorithms **but using MCTS can be beneficial in the following ways:**

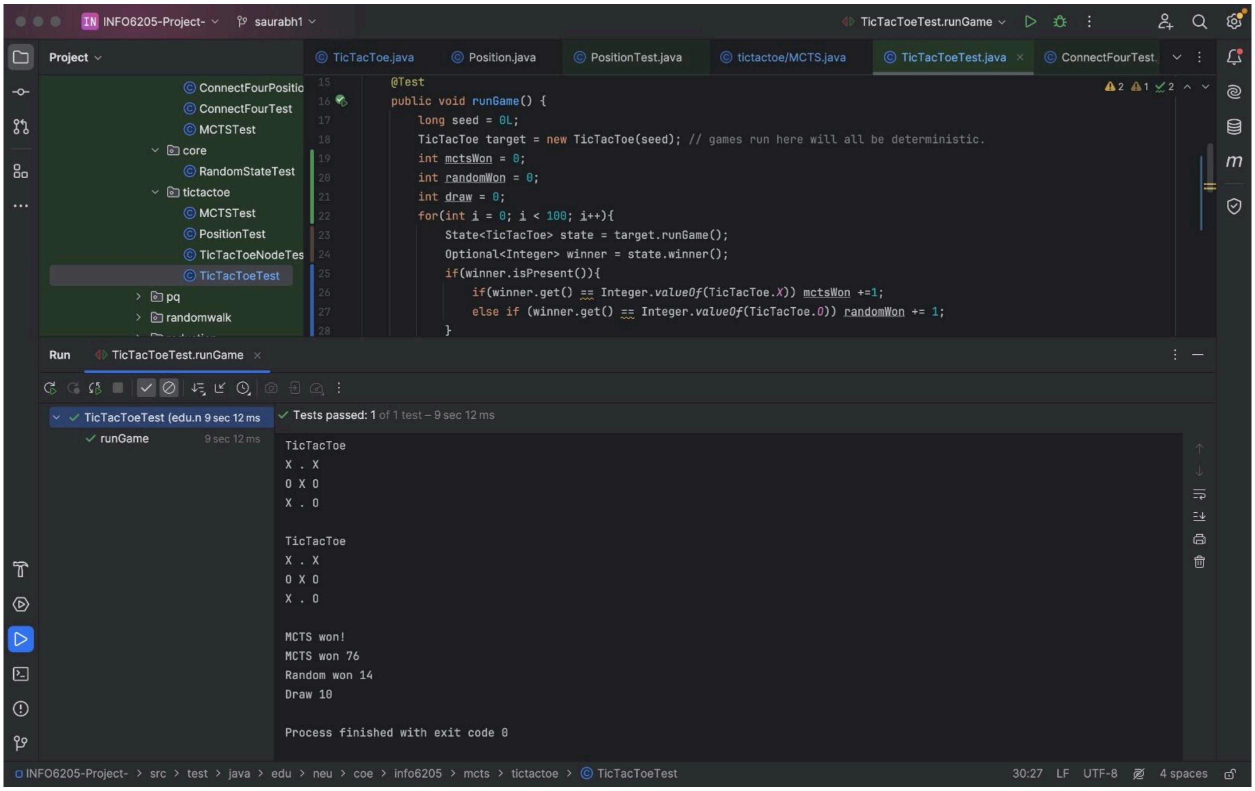
Adaptability: MCTS does not require a highly detailed evaluation function or deep domain knowledge, which makes it suitable for games with more complex strategies where evaluation functions can be hard to define.
Scalability: Although Tic-tac-toe has a small state space, MCTS's approach is beneficial for scaling up to more complex games. Using MCTS helps develop methodologies applicable to other, more complex games such as Go or Chess where the state space is much larger.
Flexibility: MCTS can easily adapt to variations of standard Tic-tac-toe, such as larger boards or games with more than two players, without significant changes to the logic of the algorithm.
Efficiency: For real-time decision-making scenarios, MCTS can operate under flexible time constraints, offering a good enough decision using the simulations conducted within the allowed time.

To prove this we ran the Tic-tac-toe game with and without MCTS by running it through different iterations. We passed the function through 10,100,1000,10000 iterations and the results as shown below, MCTS gives much higher number of wins than the game played randomly when the number of iterations are increased.



For further proof we have attached the below screenshots to demonstrate the number of wins for MCTS for various Iterations.

For 100 iterations:

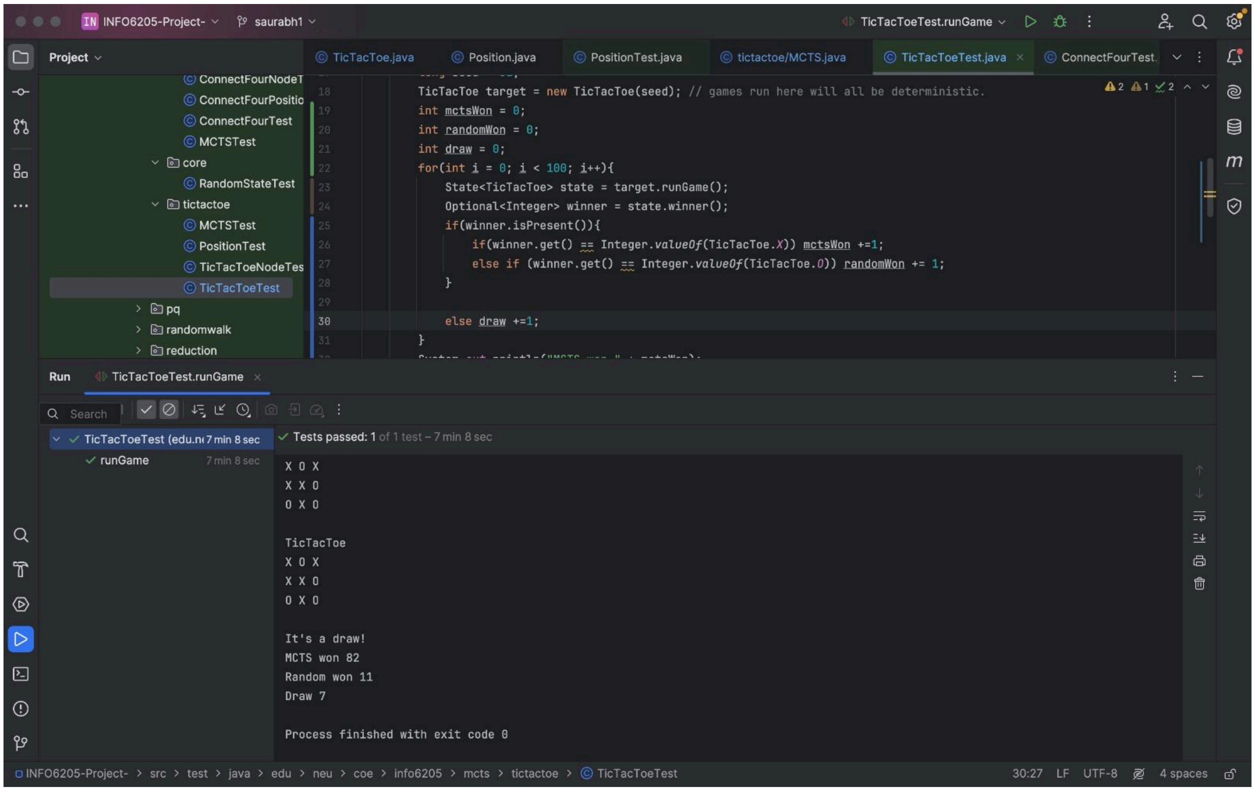


```
@Test
public void runGame() {
    long seed = 0L;
    TicTacToe target = new TicTacToe(seed); // games run here will all be deterministic.
    int mctsWon = 0;
    int randomWon = 0;
    int draw = 0;
    for(int i = 0; i < 100; i++){
        State<TicTacToe> state = target.runGame();
        Optional<Integer> winner = state.winner();
        if(winner.isPresent()){
            if(winner.get() == Integer.valueOf(TicTacToe.X)) mctsWon +=1;
            else if (winner.get() == Integer.valueOf(TicTacToe.O)) randomWon += 1;
        }
```

Tests passed: 1 of 1 test – 9 sec 12 ms

```
TicTacToe
X . X
O X O
X . O

TicTacToe
X . X
O X O
X . O

MCTS won!
MCTS won 76
Random won 14
Draw 10

Process finished with exit code 0
```

For 10000 Iterations:

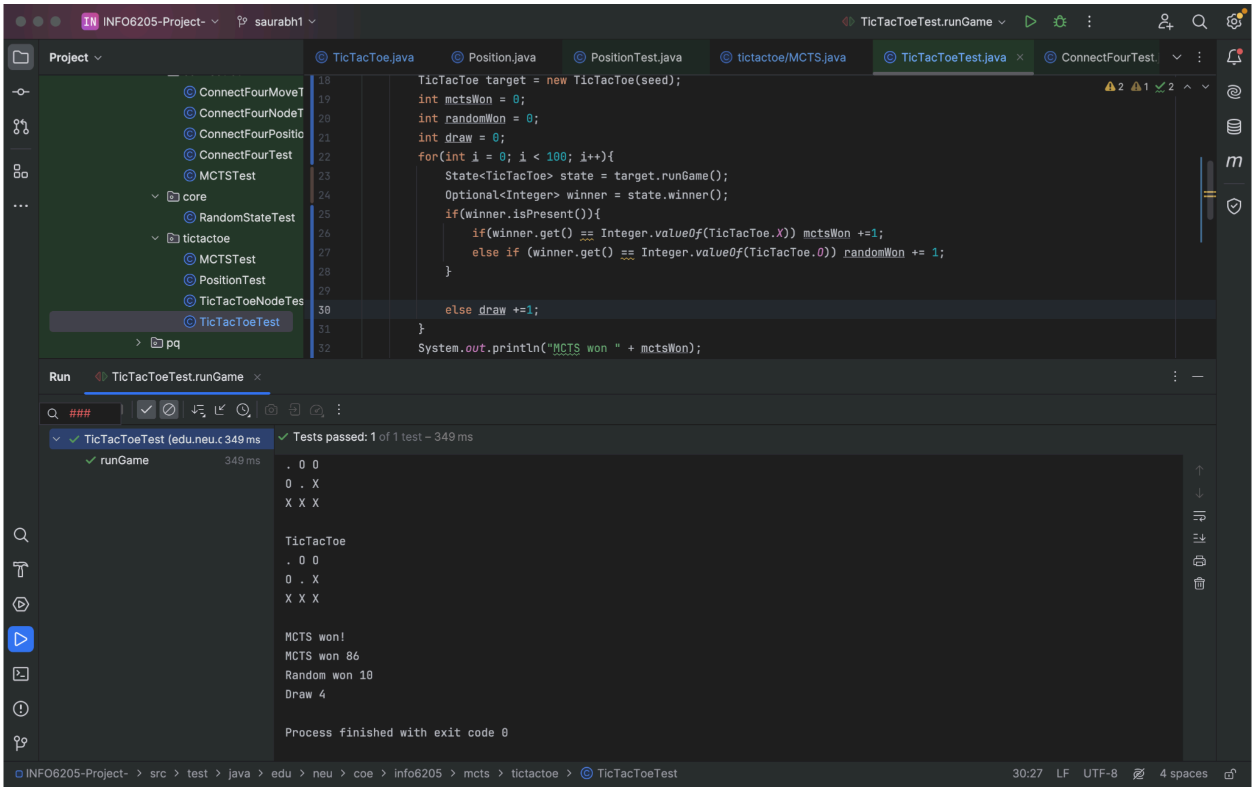


```
    TicTacToe target = new TicTacToe(seed); // games run here will all be deterministic.
    int mctsWon = 0;
    int randomWon = 0;
    int draw = 0;
    for(int i = 0; i < 100; i++){
        State<TicTacToe> state = target.runGame();
        Optional<Integer> winner = state.winner();
        if(winner.isPresent()){
            if(winner.get() == Integer.valueOf(TicTacToe.X)) mctsWon +=1;
            else if (winner.get() == Integer.valueOf(TicTacToe.O)) randomWon += 1;
        }

        else draw +=1;
    }
```

Tests passed: 1 of 1 test – 7 min 8 sec

```
X O X
X X O
O X O

TicTacToe
X O X
X X O
O X O

It's a draw!
MCTS won 82
Random won 11
Draw 7

Process finished with exit code 0
```

For 1000 iterations

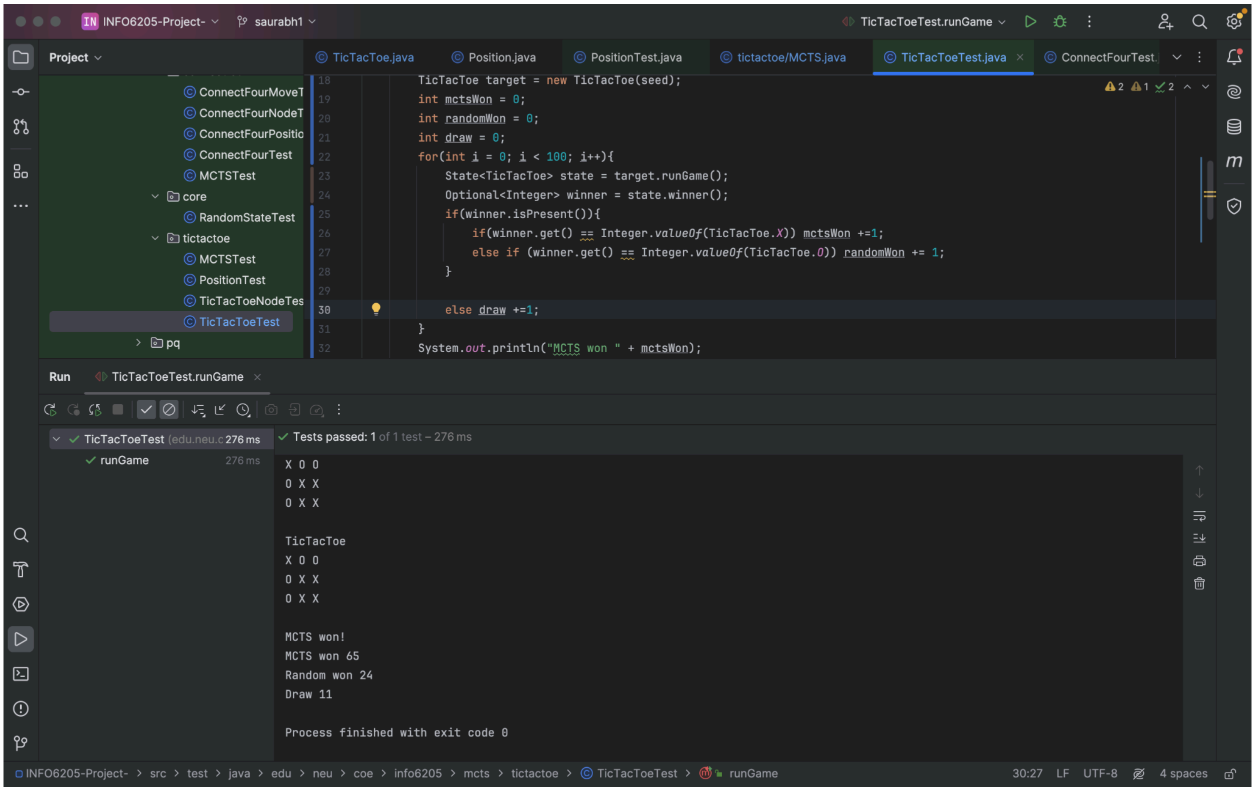


```
TicTacToe target = new TicTacToe(seed);
int mctsWon = 0;
int randomWon = 0;
int draw = 0;
for(int i = 0; i < 100; i++){
    State<TicTacToe> state = target.runGame();
    Optional<Integer> winner = state.winner();
    if(winner.isPresent()){
        if(winner.get() == Integer.valueOf(TicTacToe.X)) mctsWon +=1;
        else if (winner.get() == Integer.valueOf(TicTacToe.O)) randomWon += 1;
    }

    else draw +=1;
}
System.out.println("MCTS won " + mctsWon);
```

✓ Tests passed: 1 of 1 test – 349 ms

```
. O O
O . X
X X X

TicTacToe
. O O
O . X
X X X

MCTS won!
MCTS won 86
Random won 10
Draw 4

Process finished with exit code 0
```

For 10 iterations

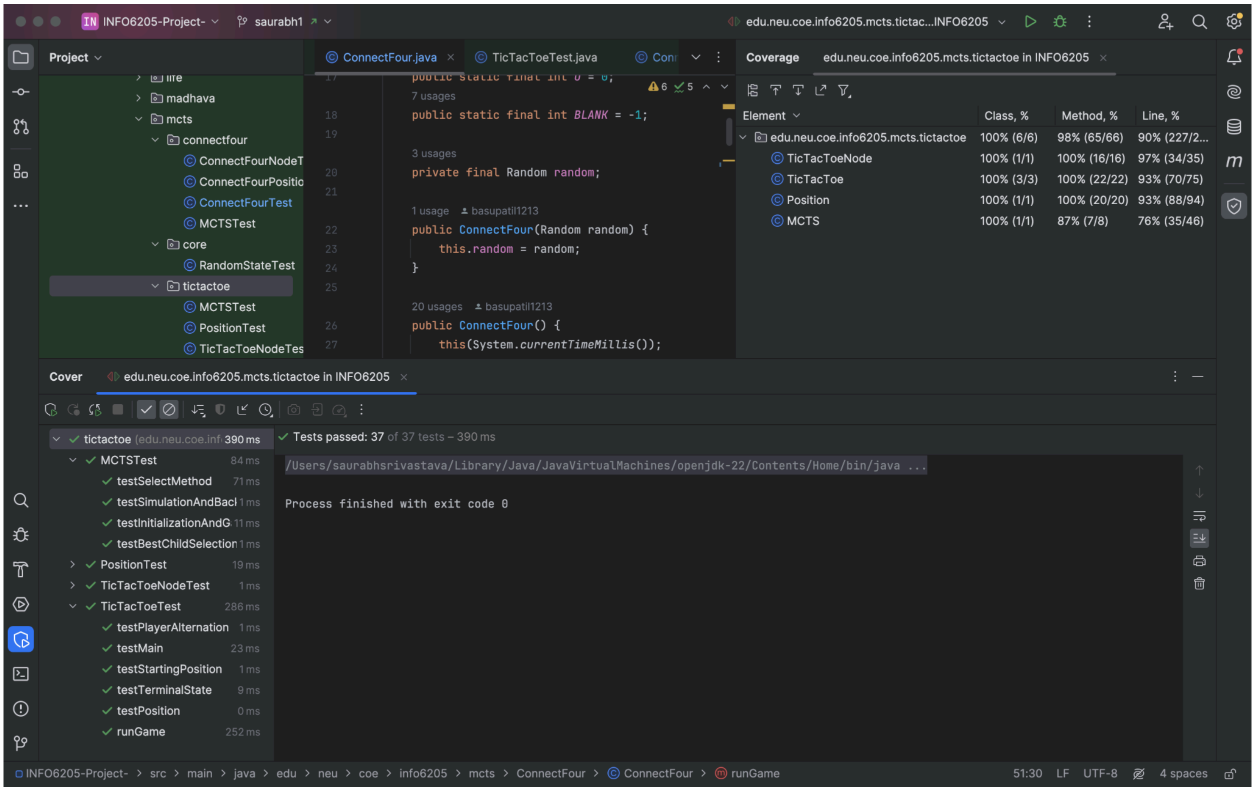


```
TicTacToe target = new TicTacToe(seed);
int mctsWon = 0;
int randomWon = 0;
int draw = 0;
for(int i = 0; i < 100; i++){
    State<TicTacToe> state = target.runGame();
    Optional<Integer> winner = state.winner();
    if(winner.isPresent()){
        if(winner.get() == Integer.valueOf(TicTacToe.X)) mctsWon +=1;
        else if (winner.get() == Integer.valueOf(TicTacToe.O)) randomWon += 1;
    }

    else draw +=1;
}
System.out.println("MCTS won " + mctsWon);
```

✓ Tests passed: 1 of 1 test – 276 ms

```
X O O
O X X
O X X

TicTacToe
X O O
O X X
O X X

MCTS won!
MCTS won 65
Random won 24
Draw 11

Process finished with exit code 0
```

We also wrote unit test cases for Tic-Tac-Toe and below are the screenshots that depict the successful running of the test cases and covering all the functions implemented.



```java
public static final int O = 0;
7 usages
public static final int BLANK = -1;

3 usages
private final Random random;

1 usage  basupatil1213
public ConnectFour(Random random) {
    this.random = random;
}

20 usages  basupatil1213
public ConnectFour() {
    this(System.currentTimeMillis());
```

Element	Class, %	Method, %	Line, %
edu.neu.coe.info6205.mcts.tictactoe	100% (6/6)	98% (65/66)	90% (227/2...
TicTacToeNode	100% (1/1)	100% (16/16)	97% (34/35)
TicTacToe	100% (3/3)	100% (22/22)	93% (70/75)
Position	100% (1/1)	100% (20/20)	93% (88/94)
MCTS	100% (1/1)	87% (7/8)	76% (35/46)

```
✓ tictactoe (edu.neu.coe.inf 390 ms
  ✓ MCTSTest                  84 ms
    ✓ testSelectMethod        71 ms
    ✓ testSimulationAndBacl    1 ms
    ✓ testInitializationAndG  11 ms
    ✓ testBestChildSelection   1 ms
  ✓ PositionTest              19 ms
  ✓ TicTacToeNodeTest          1 ms
  ✓ TicTacToeTest            286 ms
    ✓ testPlayerAlternation    1 ms
    ✓ testMain                23 ms
    ✓ testStartingPosition     1 ms
    ✓ testTerminalState        9 ms
    ✓ testPosition             0 ms
    ✓ runGame                252 ms
```

```
✓ Tests passed: 37 of 37 tests – 390 ms

/Users/saurabhsrivastava/Library/Java/JavaVirtualMachines/openjdk-22/Contents/Home/bin/java ...

Process finished with exit code 0
```

III.Implementing Connect4

The Game we have chosen is Connect4. This game is played on a vertically suspended 7x6 grid where two players take turns dropping colored discs into one of the seven columns from the top. Each column can hold up to six discs, filling from the bottom up. The objective is to be the first to form a horizontal, vertical, or diagonal line of four of one's own discs.

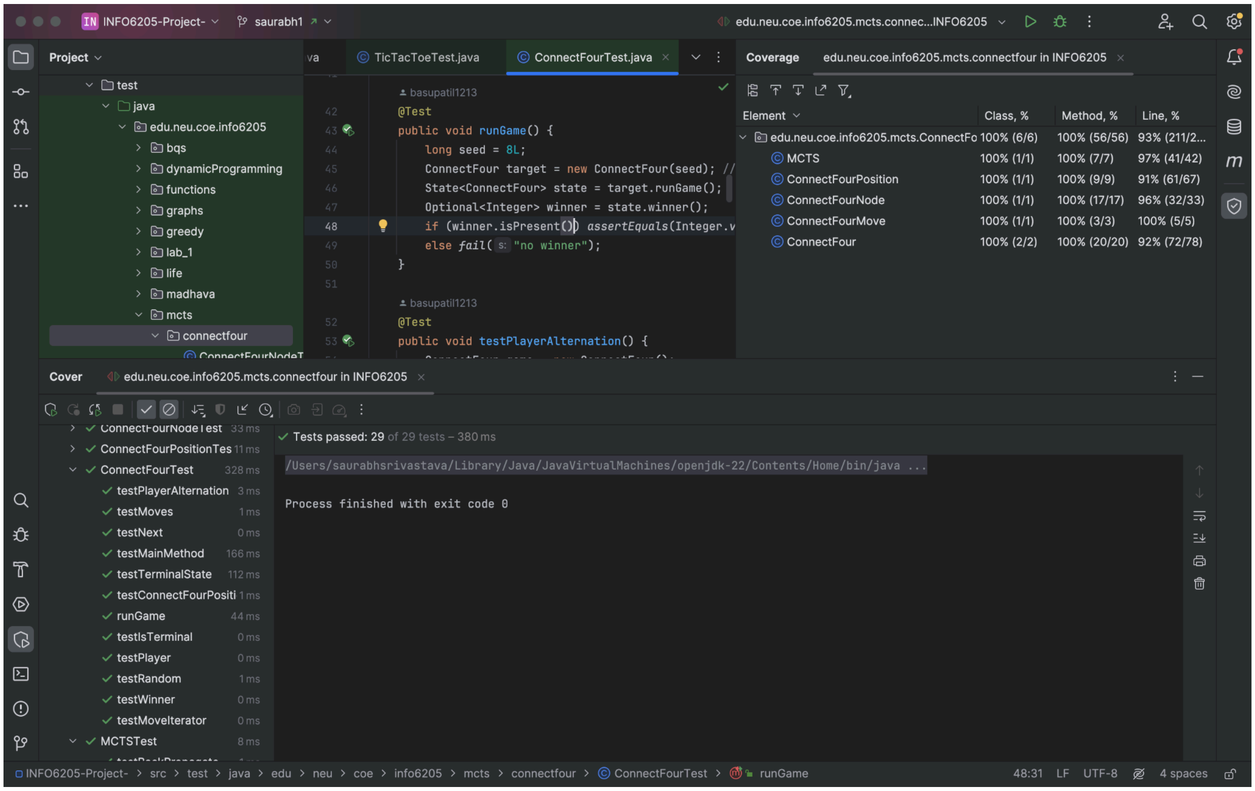
In our Connect Four game implementation, the board begins empty with all cells in a neutral state, represented programmatically by the value -1. Each player alternates turns, with player one represented by a 1 (often associated with the color red) and player two by a 0 (commonly using the color yellow). The user selects a column to drop their disc into, which automatically positions itself in the lowest available row within that column.

For example, if a player chooses to drop their disc into column three and it is the first move in that column, their disc will occupy the bottom-most position. If the same column is selected again by the next player, their disc will stack above the previous one. This continues until the column is filled or until one player successfully aligns four discs in a row, which can occur horizontally, vertically, or diagonally across the grid.

Below are the key Implementation of our Connect4 code:

- **ConnectFour.java**: This file likely contains the main game logic including the board setup, turn management, and rules for valid moves. Functions here would initialize the game state and control the flow of the game until a win, loss, or draw is determined.
- **ConnectFourMove.java**: This file would define the moves available in the game. In Connect Four, a move typically involves placing a disc in one of the columns, so this class would likely represent that action, storing the column index and the player who made the move.
- **ConnectFourNode.java**: Nodes in MCTS represent game states. Each node would encapsulate a particular Connect Four game state, with references to child nodes that represent potential future moves.
- **ConnectFourPosition.java**: This class would manage the board's state, handling the placement of discs, checking for wins or draws, and generating a list of all possible moves from the current state.
- **MCTS.java**: The MCTS algorithm's core is implemented here, likely with methods to perform the four phases of MCTS: selection, expansion, simulation, and backpropagation. Selection would choose which node to explore, expansion would add a new node to the tree, simulation would play out random games from the new node's state, and backpropagation would update the nodes with the results of the simulation to inform better decision-making in future iterations.

We could successfully run the written code for Connect4 code and below are the screenshots:
We have written several **unit tests cases for our Connect4** game and below are the screenshots of the successful passing of the Unit Tests:


INFO6205-Project-
saurabh1
edu.neu.coe.info6205.mcts.connec...INFO6205
Project
test
java
edu.neu.coe.info6205
bqs
dynamicProgramming
functions
graphs
greedy
lab_1
life
madhava
mcts
connectfour
TicTacToeTest.java
ConnectFourTest.java
Coverage
edu.neu.coe.info6205.mcts.connectfour in INFO6205
Cover
edu.neu.coe.info6205.mcts.connectfour in INFO6205
INFO6205-Project- > src > test > java > edu > neu > coe > info6205 > mcts > connectfour > ConnectFourTest > runGame
48:31 LF UTF-8 4 spaces

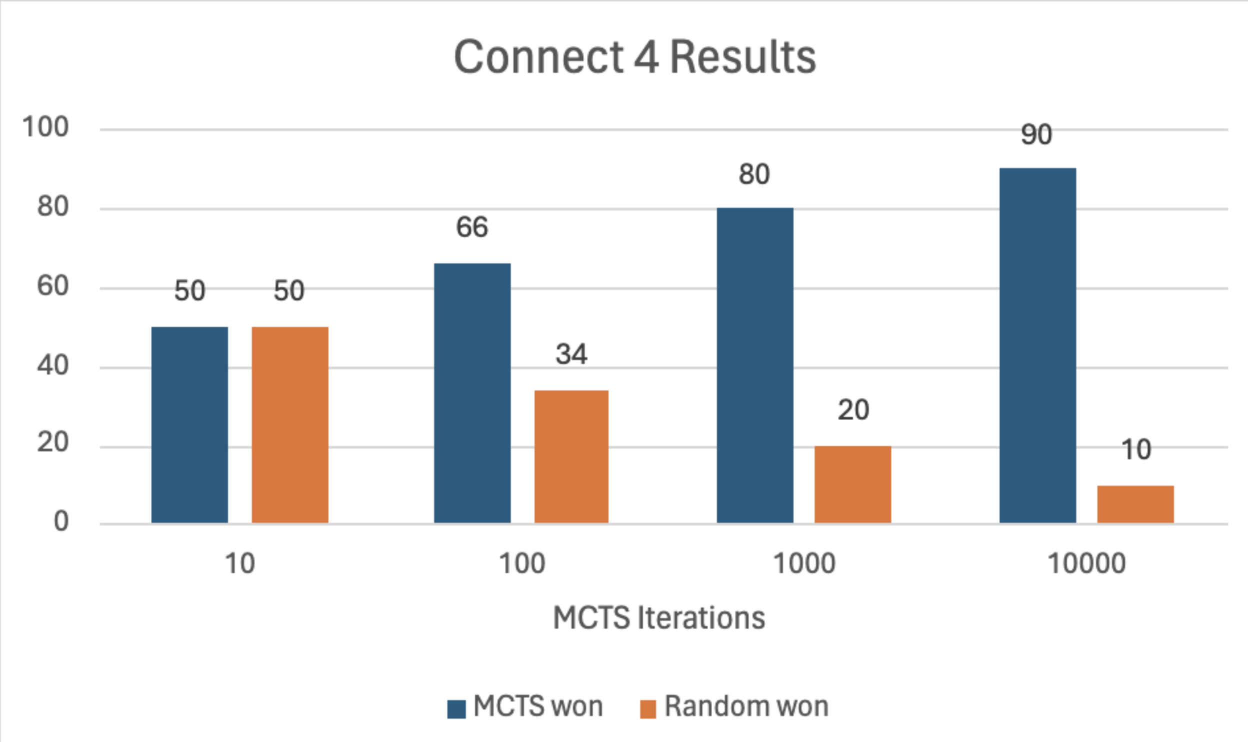
```
    basupatil1213
    @Test
    public void runGame() {
        long seed = 8L;
        ConnectFour target = new ConnectFour(seed); //
        State<ConnectFour> state = target.runGame();
        Optional<Integer> winner = state.winner();
        if (winner.isPresent()) assertEquals(Integer.v
        else fail( s: "no winner");
    }

    basupatil1213
    @Test
    public void testPlayerAlternation() {
```

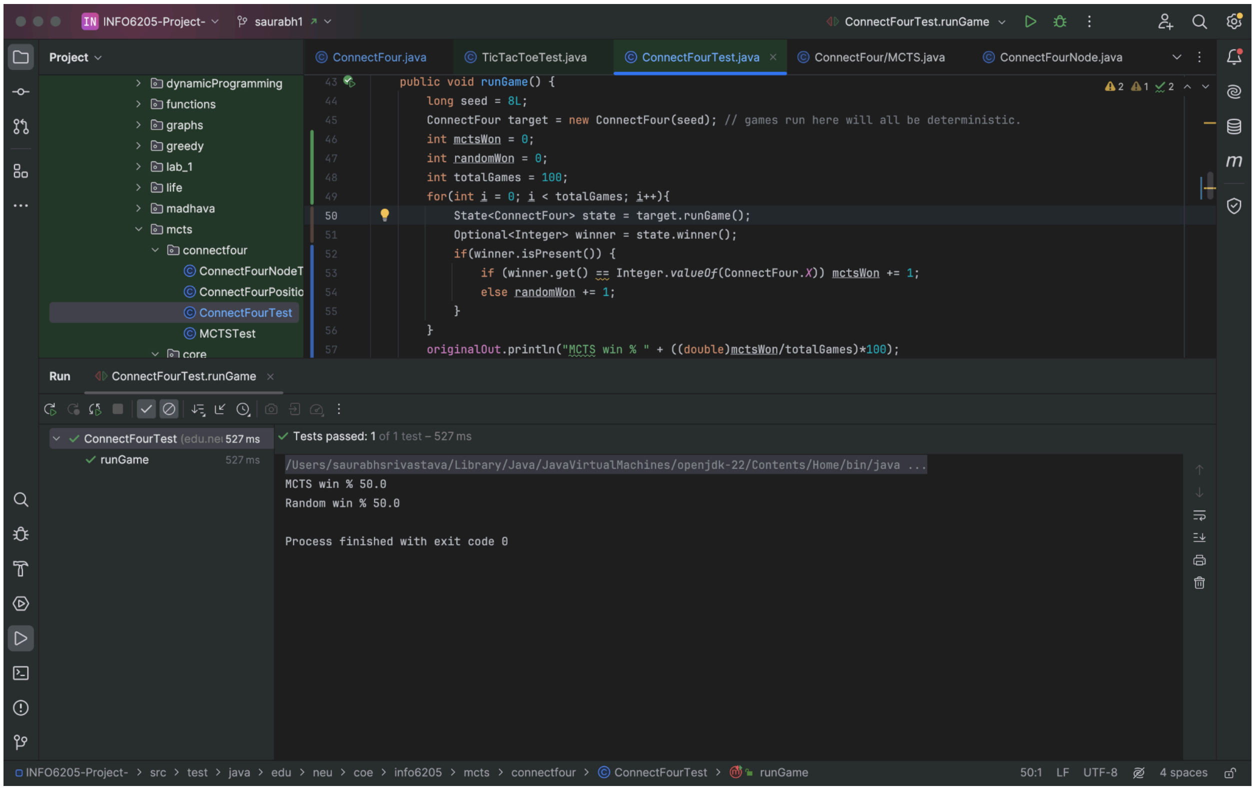
Element	Class, %	Method, %	Line, %
edu.neu.coe.info6205.mcts.ConnectFo	100% (6/6)	100% (56/56)	93% (211/2...
MCTS	100% (1/1)	100% (7/7)	97% (41/42)
ConnectFourPosition	100% (1/1)	100% (9/9)	91% (61/67)
ConnectFourNode	100% (1/1)	100% (17/17)	96% (32/33)
ConnectFourMove	100% (1/1)	100% (3/3)	100% (5/5)
ConnectFour	100% (2/2)	100% (20/20)	92% (72/78)

```
> ✓ ConnectFourNodeTest        33 ms
> ✓ ConnectFourPositionTes     11 ms
v ✓ ConnectFourTest           328 ms
    ✓ testPlayerAlternation     3 ms
    ✓ testMoves                 1 ms
    ✓ testNext                  0 ms
    ✓ testMainMethod          166 ms
    ✓ testTerminalState       112 ms
    ✓ testConnectFourPositi     1 ms
    ✓ runGame                  44 ms
    ✓ testIsTerminal            0 ms
    ✓ testPlayer                0 ms
    ✓ testRandom                1 ms
    ✓ testWinner                0 ms
    ✓ testMoveIterator          0 ms
v ✓ MCTSTest                    8 ms
```

```
✓ Tests passed: 29 of 29 tests – 380 ms

/Users/saurabhsrivastava/Library/Java/JavaVirtualMachines/openjdk-22/Contents/Home/bin/java ...

Process finished with exit code 0
```

On running Connect4 with different iterations, below are the results:
When iterations were 10 50% of the time random won and 50% of the time MCTS won, but as the iterations increased there was a significant increase in the number of wins of MCTS when compared to Random.

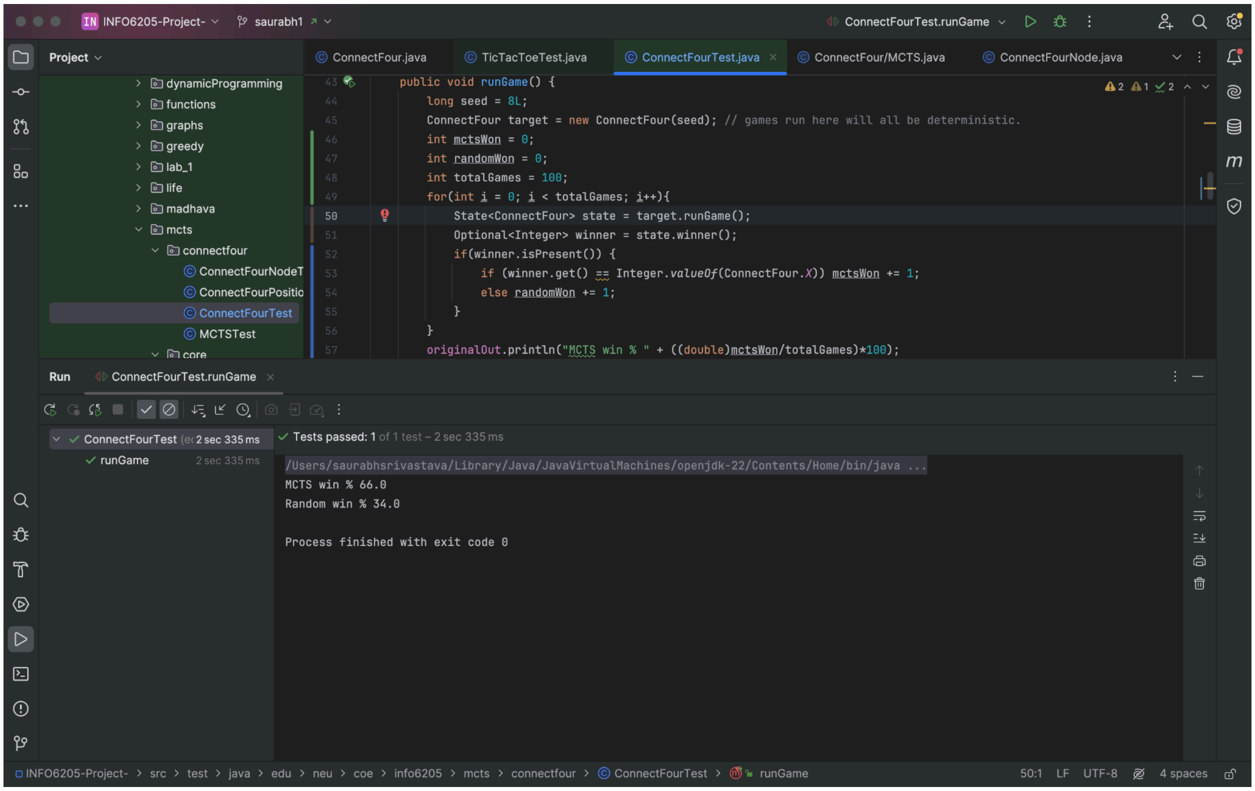




```java
public void runGame() {
    long seed = 8L;
    ConnectFour target = new ConnectFour(seed); // games run here will all be deterministic.
    int mctsWon = 0;
    int randomWon = 0;
    int totalGames = 100;
    for(int i = 0; i < totalGames; i++){
        State<ConnectFour> state = target.runGame();
        Optional<Integer> winner = state.winner();
        if(winner.isPresent()) {
            if (winner.get() == Integer.valueOf(ConnectFour.X)) mctsWon += 1;
            else randomWon += 1;
        }
    }
    originalOut.println("MCTS win % " + ((double)mctsWon/totalGames)*100);
```

✓ Tests passed: 1 of 1 test – 527 ms

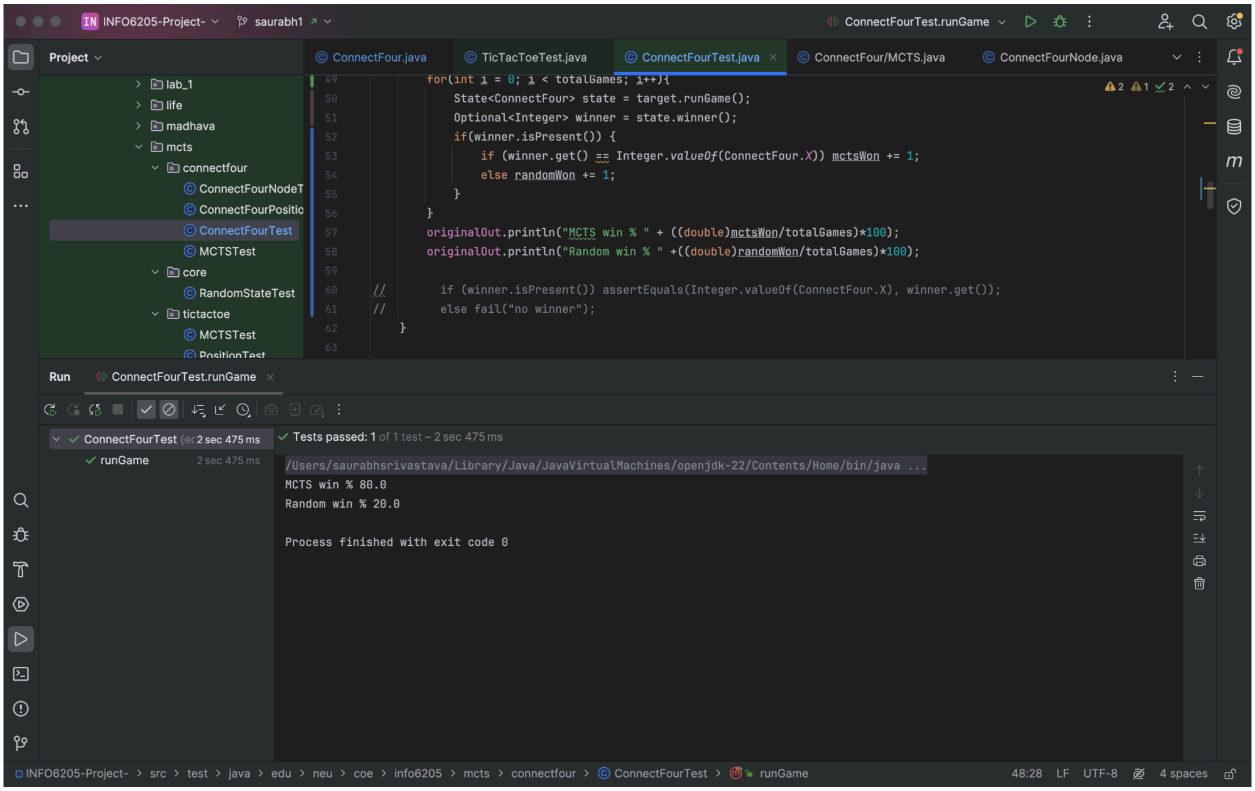
```
/Users/saurabhsrivastava/Library/Java/JavaVirtualMachines/openjdk-22/Contents/Home/bin/java ...
MCTS win % 50.0
Random win % 50.0

Process finished with exit code 0
```



```java
public void runGame() {
    long seed = 8L;
    ConnectFour target = new ConnectFour(seed); // games run here will all be deterministic.
    int mctsWon = 0;
    int randomWon = 0;
    int totalGames = 100;
    for(int i = 0; i < totalGames; i++){
        State<ConnectFour> state = target.runGame();
        Optional<Integer> winner = state.winner();
        if(winner.isPresent()) {
            if (winner.get() == Integer.valueOf(ConnectFour.X)) mctsWon += 1;
            else randomWon += 1;
        }
    }
    originalOut.println("MCTS win % " + ((double)mctsWon/totalGames)*100);
```

✓ Tests passed: 1 of 1 test – 2 sec 335 ms

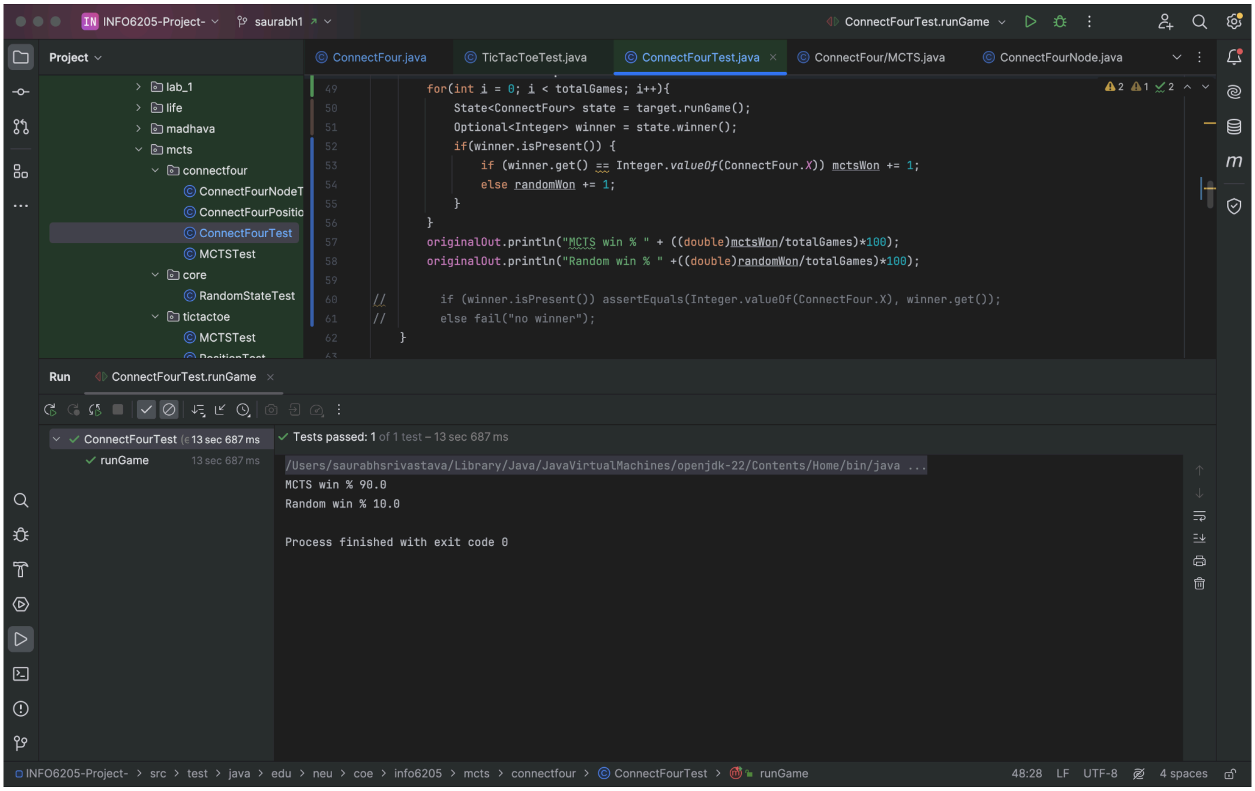
```
/Users/saurabhsrivastava/Library/Java/JavaVirtualMachines/openjdk-22/Contents/Home/bin/java ...
MCTS win % 66.0
Random win % 34.0

Process finished with exit code 0
```



```
        for(int i = 0; i < totalGames; i++){
            State<ConnectFour> state = target.runGame();
            Optional<Integer> winner = state.winner();
            if(winner.isPresent()) {
                if (winner.get() == Integer.valueOf(ConnectFour.X)) mctsWon += 1;
                else randomWon += 1;
            }
        }
        originalOut.println("MCTS win % " + ((double)mctsWon/totalGames)*100);
        originalOut.println("Random win % " +((double)randomWon/totalGames)*100);

//        if (winner.isPresent()) assertEquals(Integer.valueOf(ConnectFour.X), winner.get());
//        else fail("no winner");
    }
```

✓ Tests passed: 1 of 1 test – 2 sec 475 ms

```
/Users/saurabhsrivastava/Library/Java/JavaVirtualMachines/openjdk-22/Contents/Home/bin/java ...
MCTS win % 80.0
Random win % 20.0

Process finished with exit code 0
```

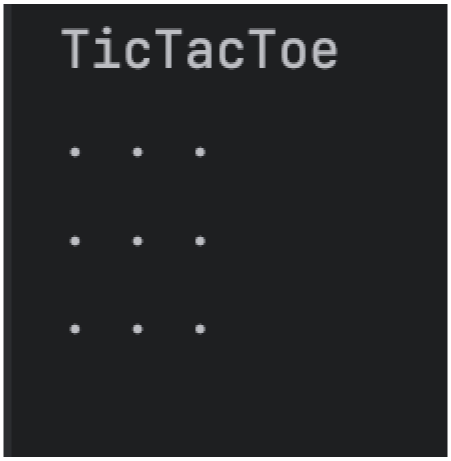


```
        for(int i = 0; i < totalGames; i++){
            State<ConnectFour> state = target.runGame();
            Optional<Integer> winner = state.winner();
            if(winner.isPresent()) {
                if (winner.get() == Integer.valueOf(ConnectFour.X)) mctsWon += 1;
                else randomWon += 1;
            }
        }
        originalOut.println("MCTS win % " + ((double)mctsWon/totalGames)*100);
        originalOut.println("Random win % " +((double)randomWon/totalGames)*100);

//        if (winner.isPresent()) assertEquals(Integer.valueOf(ConnectFour.X), winner.get());
//        else fail("no winner");
    }
```

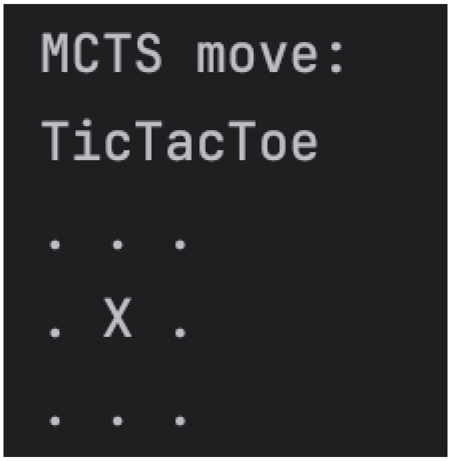
✓ Tests passed: 1 of 1 test – 13 sec 687 ms

```
/Users/saurabhsrivastava/Library/Java/JavaVirtualMachines/openjdk-22/Contents/Home/bin/java ...
MCTS win % 90.0
Random win % 10.0

Process finished with exit code 0
```

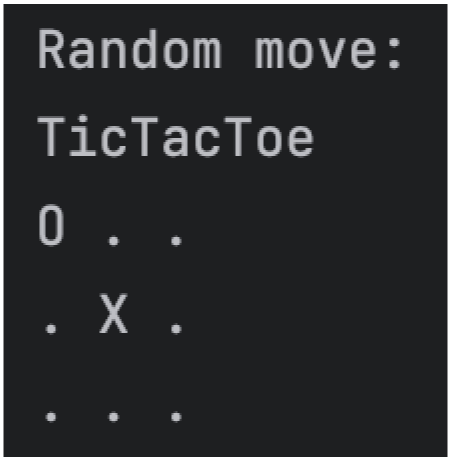
As we run the Tic-tac-toe game we have captured screenshots of the **Step-By-Step results as we proceed through each Move for the Tic-Tac-Toe Game**. Below are the screenshots for both Random Tic-tac-toe as well as the game using MCTS.



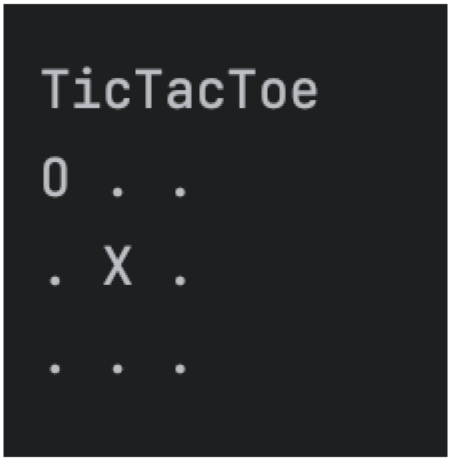
```
TicTacToe
. . .
. . .
. . .
```



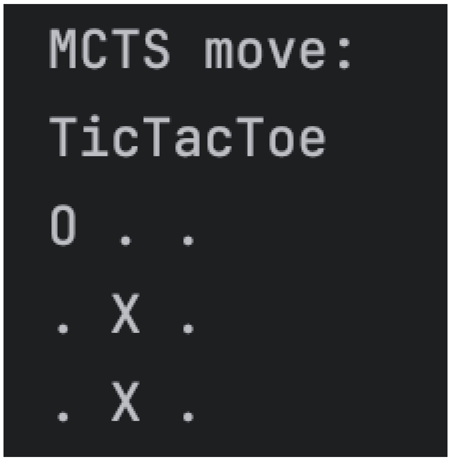
```
MCTS move:
TicTacToe
. . .
. X .
. . .
```



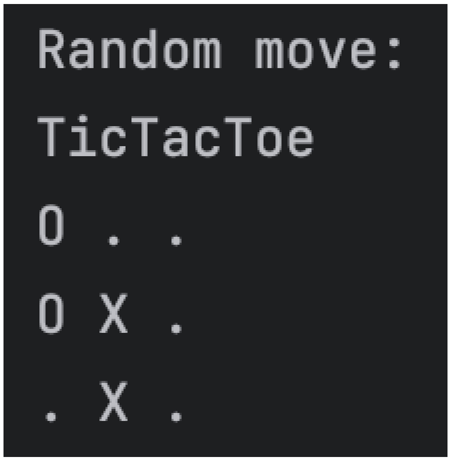
```
Random move:
TicTacToe
O . .
. X .
. . .
```



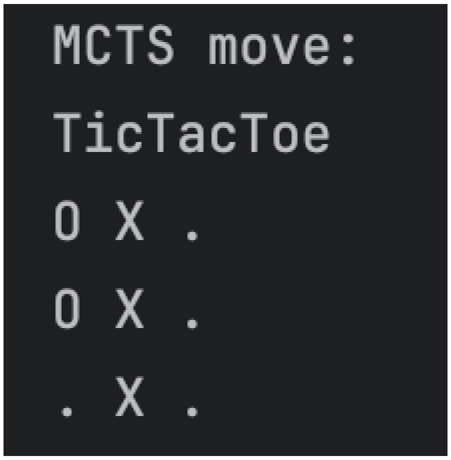
```
TicTacToe
O . .
. X .
. . .
```



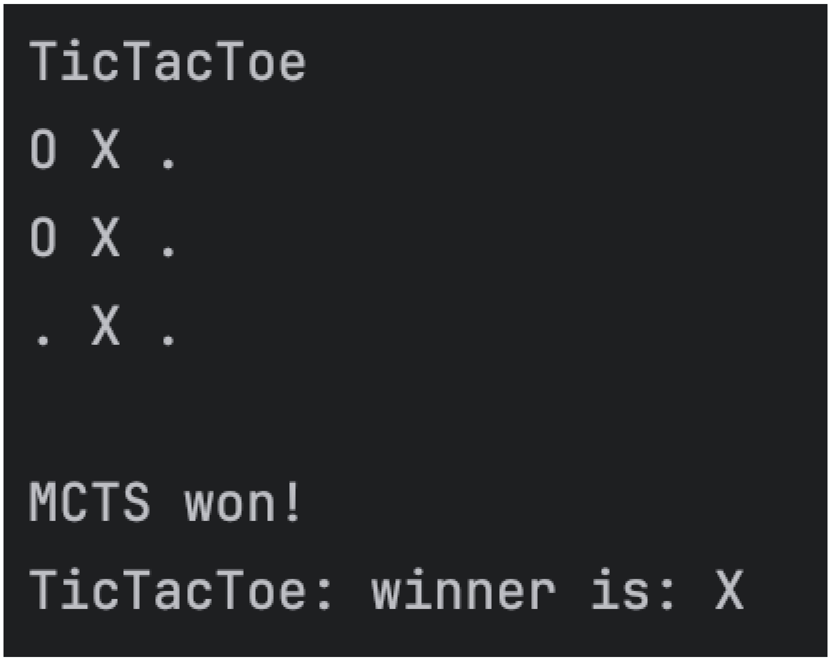
```
MCTS move:
TicTacToe
O . .
. X .
. X .
```



```
Random move:
TicTacToe
O . .
O X .
. X .
```

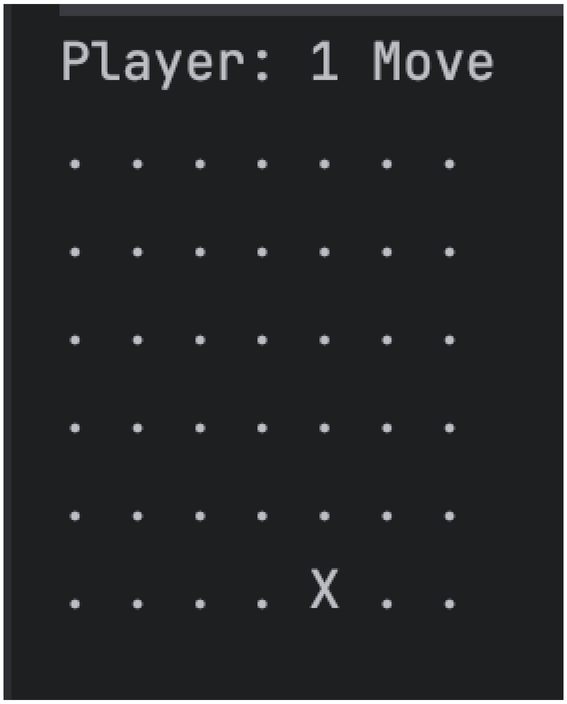


```
MCTS move:
TicTacToe
O X .
O X .
. X .
```

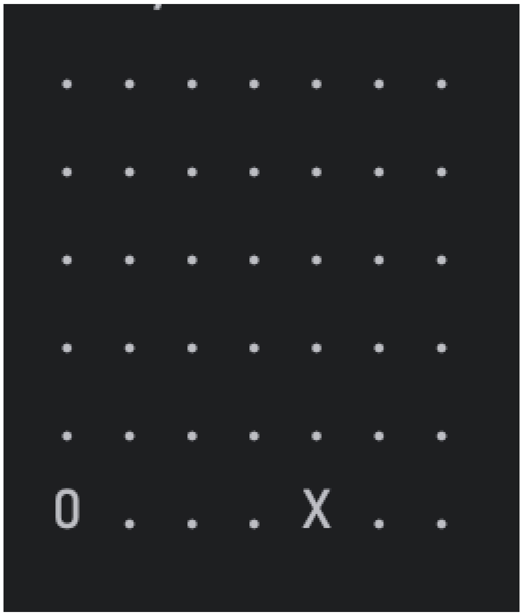


```
TicTacToe
O X .
O X .
. X .

MCTS won!
TicTacToe: winner is: X
```

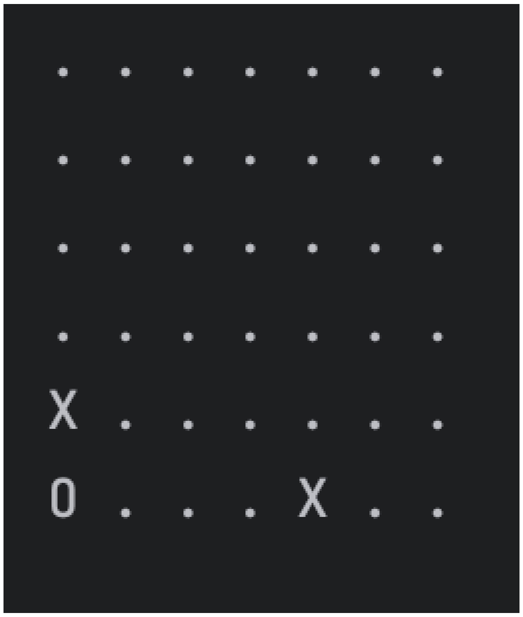
As we run the Connect4 game we have captured screenshots of the **Step-By-Step results as we proceed through each Move for the** Connect4 **Game**. Below are the screenshots for both the Random and the MCTS algorithm for Connect4 game.



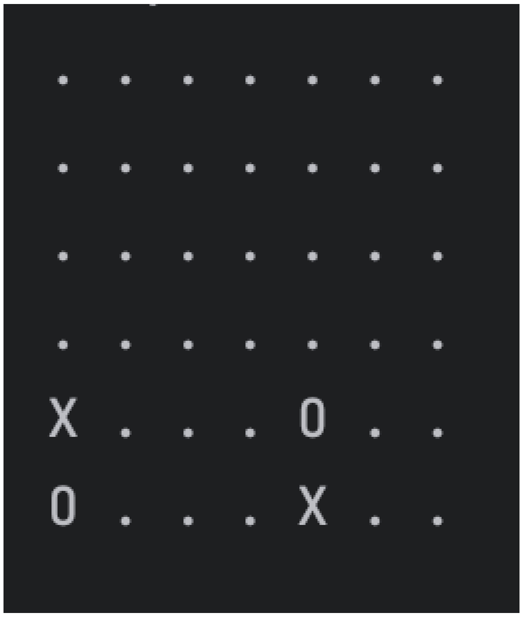
```
Player: 1 Move

. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . X . .
```



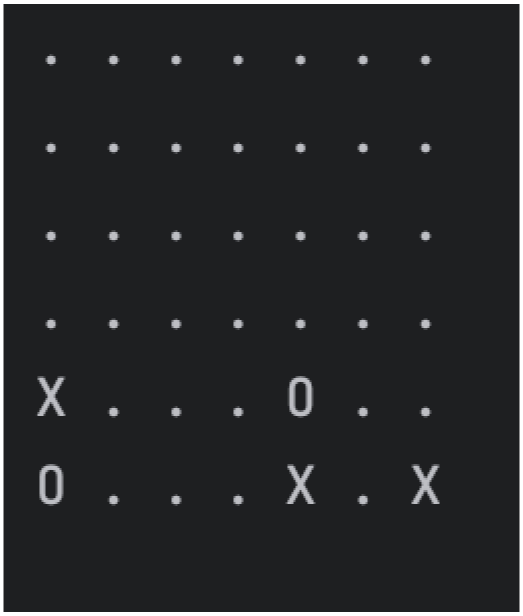
```
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
O . . . X . .
```



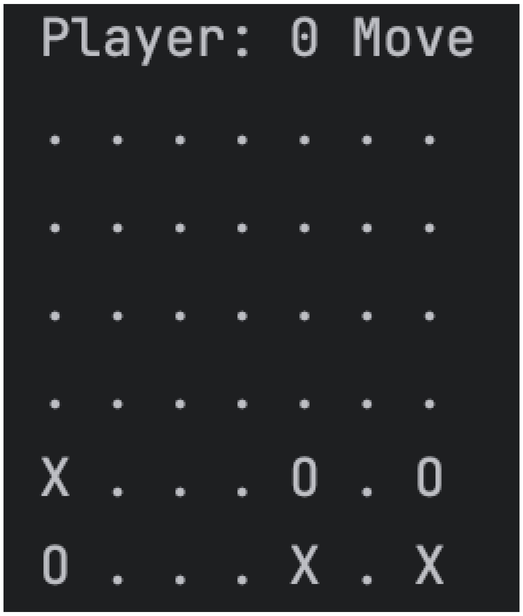
```
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
X . . . . . .
O . . . X . .
```



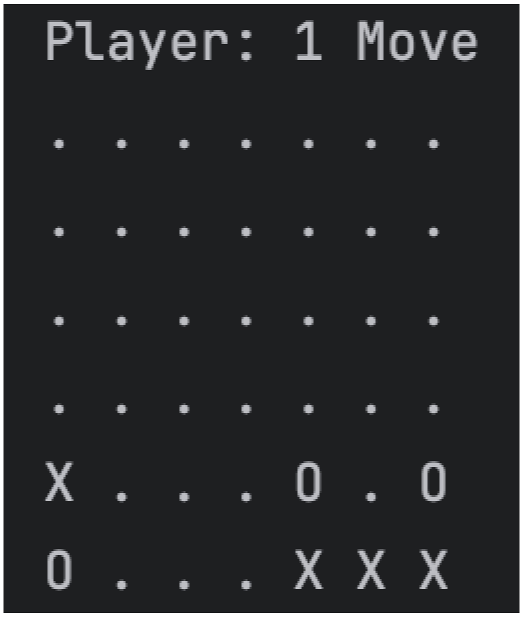
```
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
X . . . O . .
O . . . X . .
```



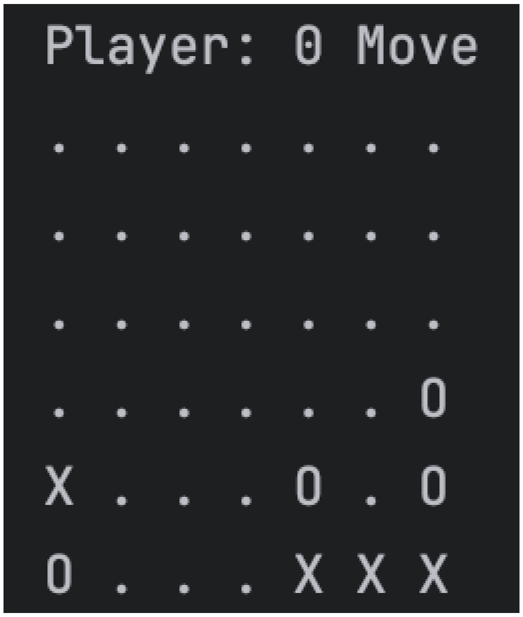
```
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
X . . . O . .
O . . . X . X
```



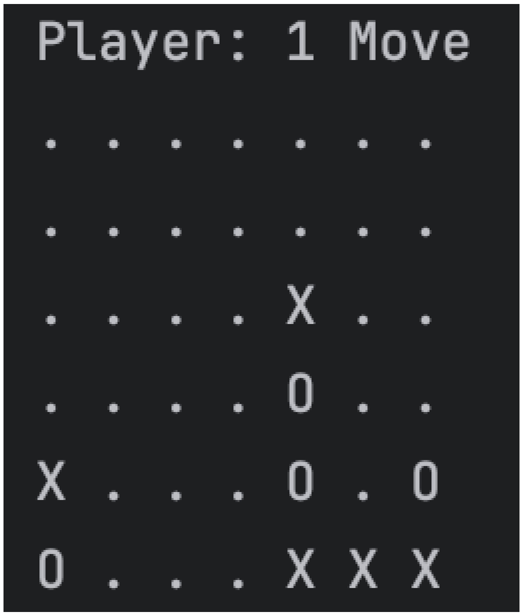
```
Player: 0 Move
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
X . . . O . O
O . . . X . X
```



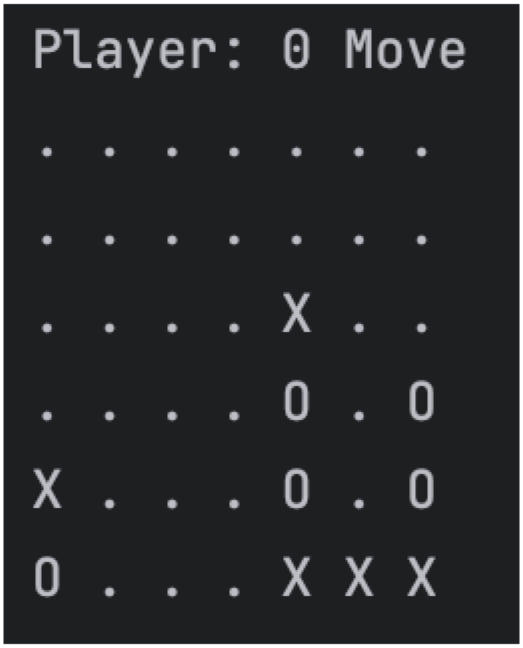
```
Player: 1 Move
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
X . . . O . O
O . . . X X X
```



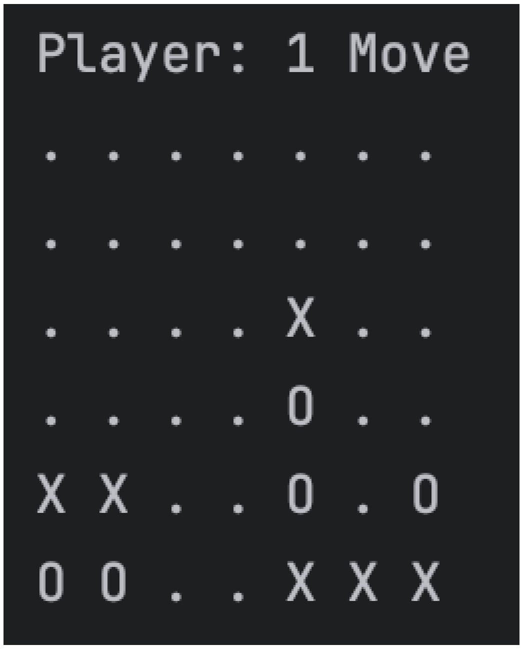
```
Player: 0 Move
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . O
X . . . O . O
O . . . X X X
```



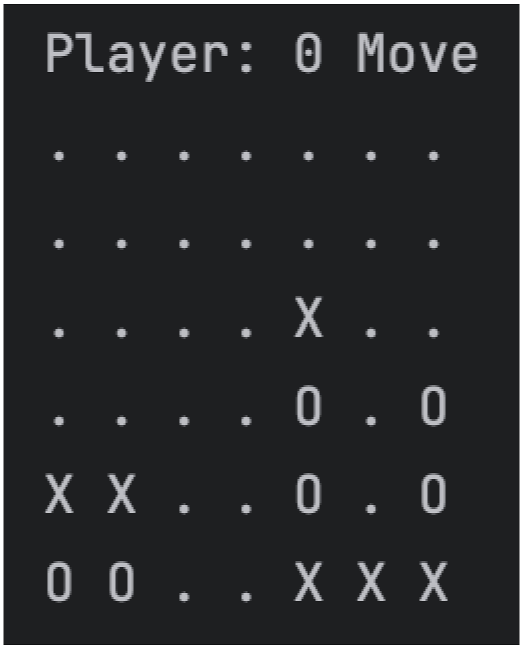
```
Player: 1 Move
. . . . . . .
. . . . . . .
. . . . X . .
. . . . O . .
X . . . O . O
O . . . X X X
```



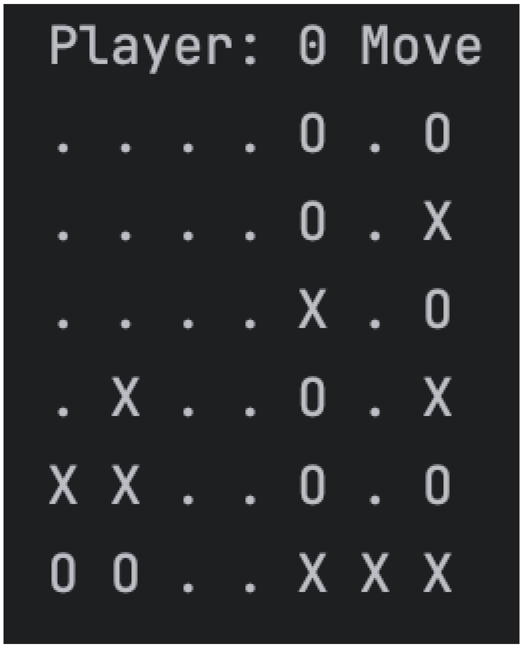
```
Player: 0 Move
. . . . . . .
. . . . . . .
. . . . X . .
. . . . O . O
X . . . O . O
O . . . X X X
```



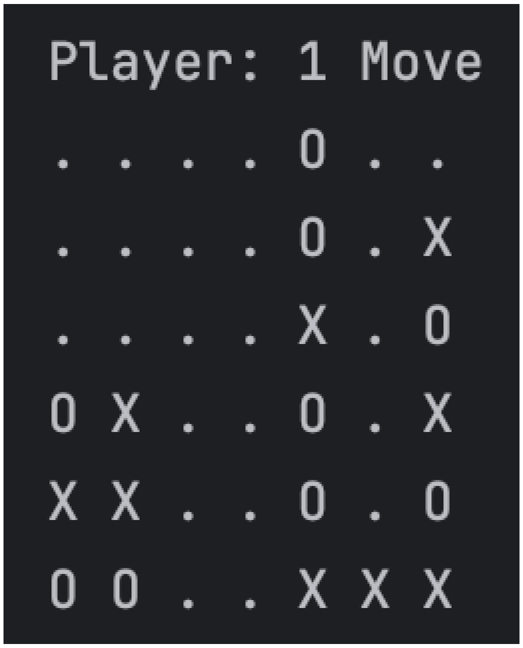
```
Player: 1 Move
. . . . . . .
. . . . . . .
. . . . X . .
. . . . O . .
X X . . O . O
O O . . X X X
```



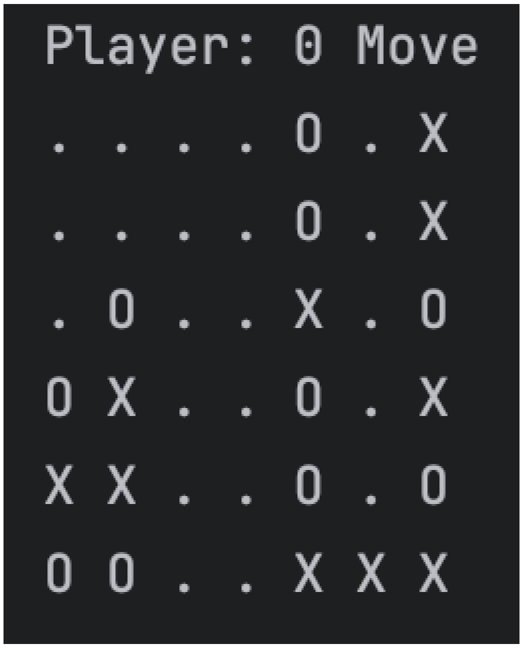
```
Player: 0 Move
. . . . . . .
. . . . . . .
. . . . X . .
. . . . O . O
X X . . O . O
O O . . X X X
```



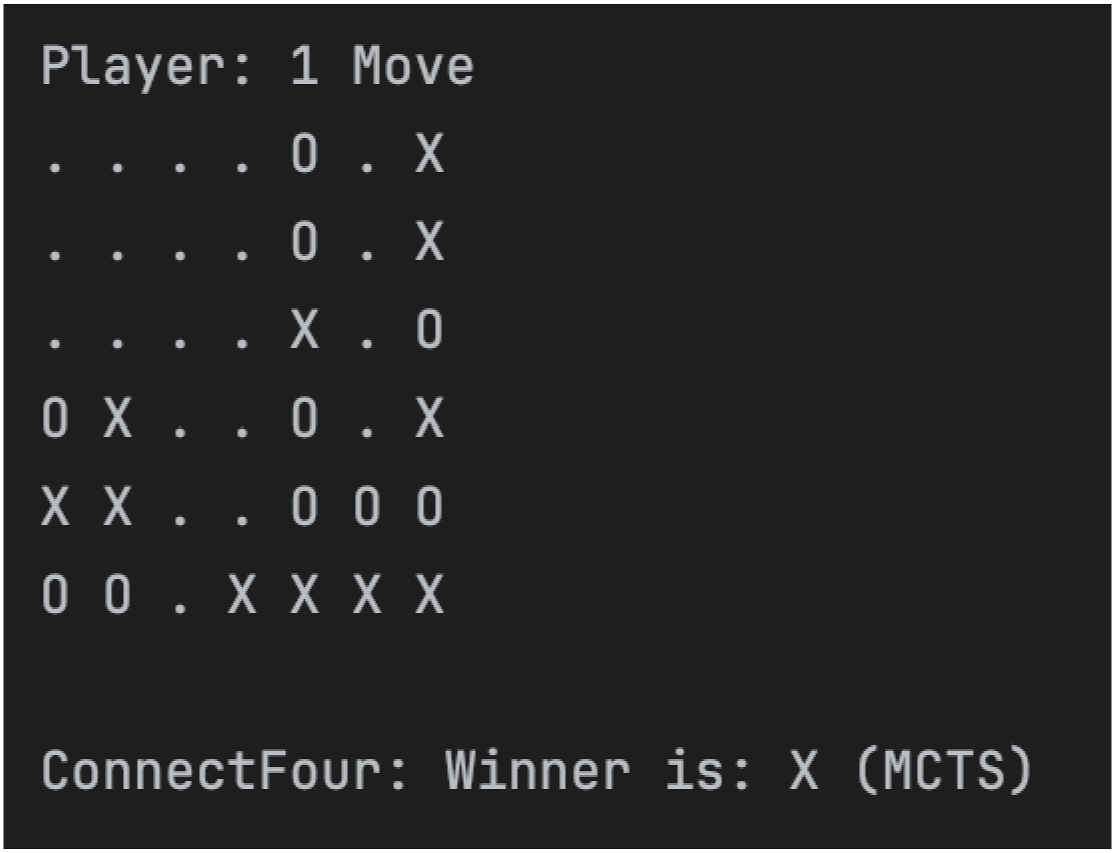
```
Player: 0 Move
. . . . O . O
. . . . O . X
. . . . X . O
. X . . O . X
X X . . O . O
O O . . X X X
```



```
Player: 1 Move
. . . . O . .
. . . . O . X
. . . . X . O
O X . . O . X
X X . . O . O
O O . . X X X
```



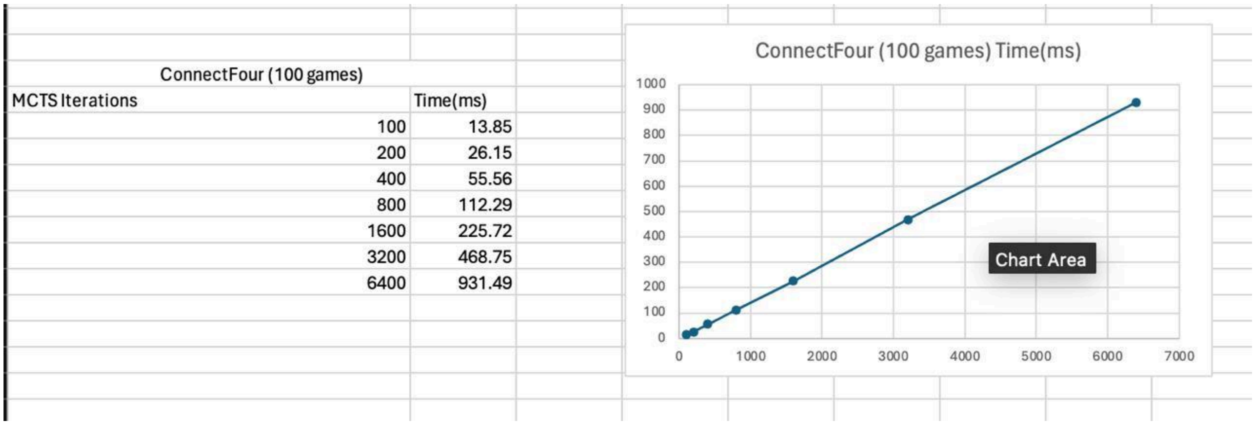
```
Player: 0 Move
. . . . O . X
. . . . O . X
. O . . X . O
O X . . O . X
X X . . O . O
O O . . X X X
```



```
Player: 1 Move
. . . . O . X
. . . . O . X
. . . . X . O
O X . . O . X
X X . . O O O
O O . X X X X


ConnectFour: Winner is: X (MCTS)
```

IV.Benchmark Results

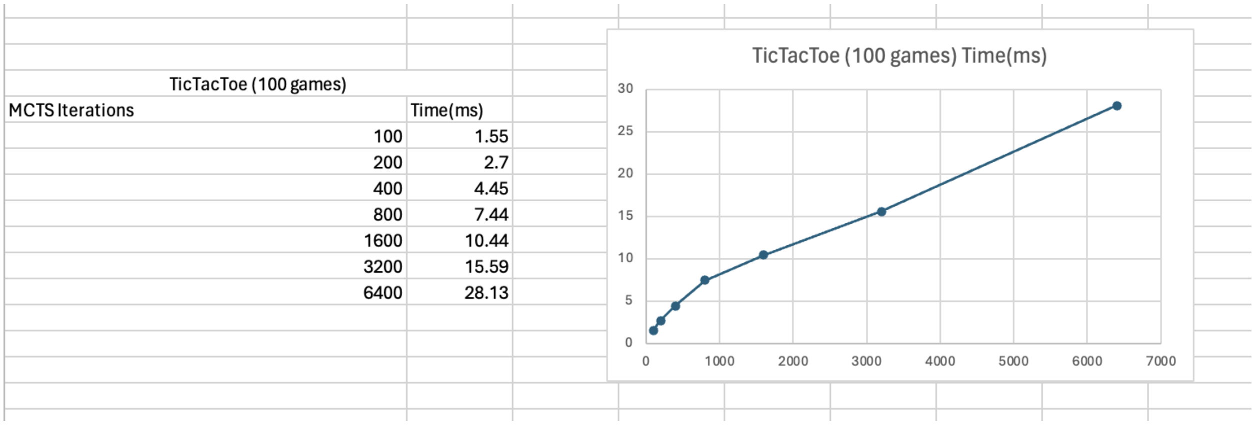
Below is a graphical implementation of the **Connect4(using MCTS)** game as we increase the number of Iterations. As the number of iterations increased the time has linearly increased.X-axis we represent the the time taken in ms and on Y-axis we represent the number of Iterations.



ConnectFour (100 games)

MCTS Iterations	Time(ms)
100	13.85
200	26.15
400	55.56
800	112.29
1600	225.72
3200	468.75
6400	931.49

Below is a graphical implementation of the **Tic-Tac-Toe game(using MCTS)** game as we increase the number of Iterations . X-axis we represent the time taken in ms and on Y-axis we represent the number of Iterations.



TicTacToe (100 games)

MCTS Iterations	Time(ms)
100	1.55
200	2.7
400	4.45
800	7.44
1600	10.44
3200	15.59
6400	28.13

V.CONCLUSION

In this project, we explored the applicability and robustness of the Monte Carlo Tree Search (MCTS) as a decision-making algorithm in the domain of two-player games, with a particular focus on Connect Four. We began with an understanding of MCTS by applying it to a sample game of Tic-Tac-Toe, which served as a preliminary step toward grasping the fundamental mechanics of the algorithm. Progressing further, we ventured into a more complex domain by implementing MCTS for Connect Four—a game that not only demands strategic foresight but also requires dynamic evaluation of the game state as the grid is larger and offers a higher degree of freedom for each move. Connect Four introduces unique challenges that were not present in the simplistic Tic-Tac-Toe game, such as the vertical accumulation of tokens and the need to align four consecutive tokens either horizontally, vertically, or diagonally. Unlike Tic-Tac-Toe, which rarely requires a search depth beyond the next few moves due to its limited board size, Connect Four's larger grid size and increased win condition necessitate deeper foresight and evaluation, making it an ideal candidate for MCTS.

Based on the Graphs presented above the results are evident as

In conclusion, the utilization of MCTS in Connect Four not only enhanced the AI's capability to handle the intricate decision-making process inherent to the game but also cemented the versatility and power of MCTS in advancing AI game-playing strategies across various game types.