Program Structures and Algorithms

Spring 2024

Final Project

Monte Carlo Project Simulation

Members: Abhinav Gangadharan (002889956),

Tarun Angrish (002807094)

GITHUB LINK:

https://github.com/tarunangrish-neu/INFO6205-MCTS-Final-Project/

Introduction:

This is an attempt to execute Monte Carlo simulations on board games. It is a type of simulation that relies on repeated random sampling and statistical analysis to compute the results. In this report, we will briefly describe the nature and relevance of Monte Carlo simulation, the way to perform these simulations and analyze results, and the underlying mathematical techniques required for performing these simulations.

Introduction To Tic-Tac-Toe:

Tic-tac-toe, also known as noughts and crosses, is a classic pencil-and-paper game played on a 3x3 grid. Two players, typically represented by X and O markers, take turns placing their symbol in an empty cell of the grid. The objective is to form a horizontal, vertical, or diagonal line of three of your symbols.
The game starts with an empty grid, and players alternate turns until one player achieves the winning condition or the grid fills up, resulting in a draw. Players aim to strategically place their marks to block their opponent's progress while seeking opportunities to create their own winning lines.
Tic-tac-toe is popular for its simplicity yet offers a surprising depth of strategy, especially when played by experienced competitors. Due to its straightforward rules and minimal equipment requirements, it's often used as a teaching tool for basic game theory and artificial intelligence concepts.
Though the game has a finite number of possible positions, its simplicity and strategic elements have made it a timeless favorite among players of all ages, often enjoyed as a quick pastime or educational exercise in logical thinking.

MCTS on Tic-Tac-Toe:
Monte Carlo Tree Simulation (MCTS) is a heuristic search algorithm used in decision processes, notably in artificial intelligence and game theory. It operates by building a tree of possible game states, where each node represents a particular state and the edges denote possible moves leading to subsequent states. Unlike traditional tree search algorithms that explore all branches exhaustively, MCTS employs a probabilistic approach, focusing its exploration on promising areas of the search space.
At each iteration, MCTS selects a node to explore based on a selection policy, usually balancing between exploration (visiting less-explored nodes) and exploitation (focusing on nodes with high potential for success). It then simulates gameplay from that node to a terminal state, typically using randomized or Monte Carlo sampling methods. The outcome of these simulations informs the evaluation of the node, which is propagated back up the tree to update the statistics of the traversed nodes.
Over multiple iterations, MCTS gradually refines its understanding of the game space, biasing its exploration towards more promising moves while continuously improving its decision-making capabilities. This makes it particularly effective in scenarios with large search spaces and imperfect information, such as complex board games like Go or chess.
It consists of 4 Phases:

1. Selection

In this initial phase, the algorithm starts with a root node and selects a child node such that it picks the node with maximum win rate. We also want to make sure that each node is given a fair chance.
The idea is to keep selecting optimal child nodes until we reach the leaf node of the tree. A good way to select such a child node is to use UCT (Upper Confidence Bound applied to trees) formula:

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln t}{n_i}}$$

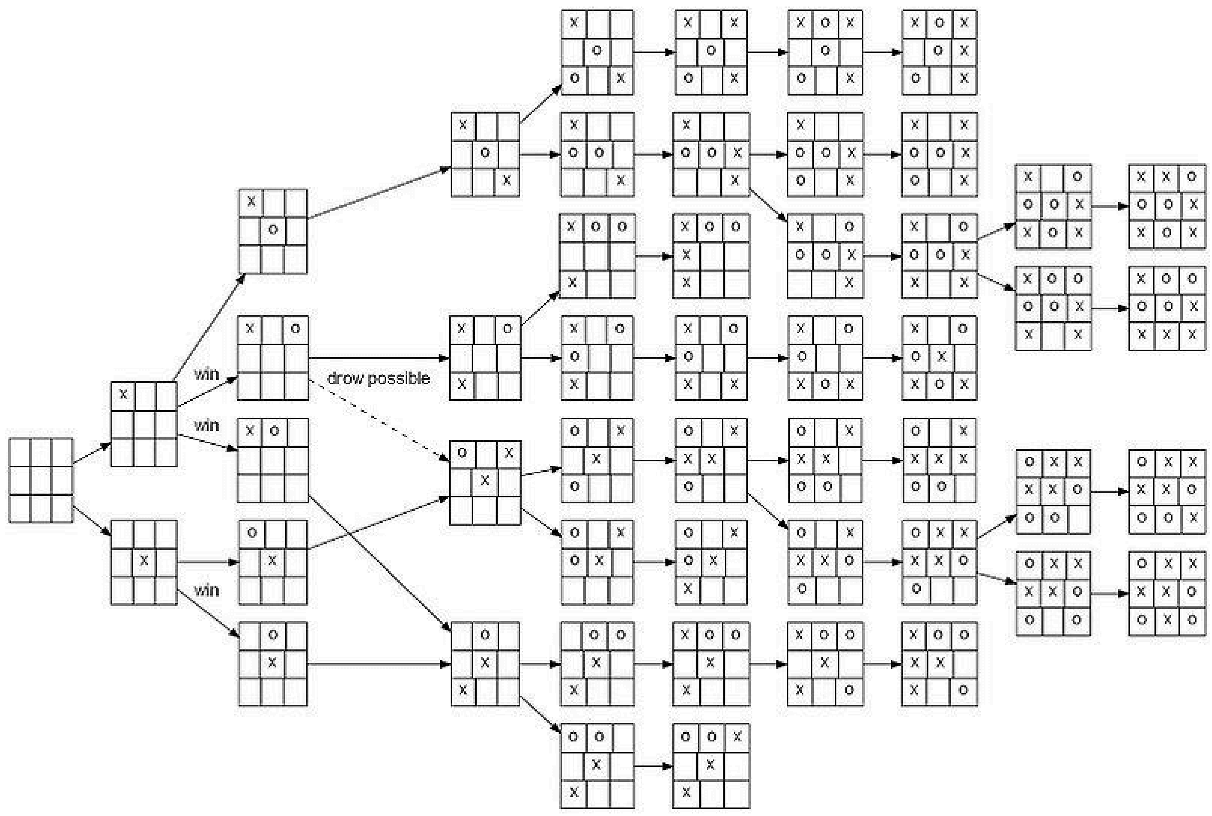
In which

- wi = number of wins after the i-th move
- ni = number of simulations after the i-th move
- c = exploration parameter (theoretically equal to √2)
- t = total number of simulations for the parent node

The formula ensures that no state will be a victim of starvation and it also plays promising branches more often than their counterparts.

2. Expansion

When it can no longer apply UCT to find the successor node, it expands the game tree by appending all possible states from the leaf node.



3. Simulation

After Expansion, the algorithm picks a child node arbitrarily, and it simulates a randomized game from selected node until it reaches the resulting state of the game. If nodes are picked randomly or semi-randomly during the play out, it is called light play out. You can also opt for heavy play out by writing quality heuristics or evaluation functions.

4. Backpropagation

This is also known as an update phase. Once the algorithm reaches the end of the game, it evaluates the state to figure out which player has won. It traverses upwards to the root and increments visit score for all visited nodes. It also updates win score for each node if the player for that position has won the playout.
MCTS keeps repeating these four phases until some fixed number of iterations or some fixed amount of time.
In this approach, we estimate winning score for each node based on random moves. So higher the number of iterations, more reliable the estimate becomes. The algorithm estimates will be less accurate at the start of a search and keep improving after sufficient amount of time. Again it solely depends on the type of the problem.
Our Implementation is explained in brief below:

MCTS Class →

This class represents the Monte Carlo Tree Search algorithm.

- main method:

    - Initializes the Tic-Tac-Toe game and the root node of the search tree.
    - Runs the MCTS algorithm for a specified number of iterations.
    - Prints the recommended move based on the best child node found by MCTS.
- run method:
    - Executes the MCTS algorithm for the specified number of iterations.
    - In each iteration, it performs selection, expansion, simulation, and backpropagation.
- select method:
    - Selects a node in the tree for expansion.
    - It traverses down the tree by selecting child nodes based on a selection policy until it reaches a leaf node.
- bestChild method:
    - Selects the child node with the highest UCT (Upper Confidence Bound for Trees) value.
    - It calculates the UCT value for each child node and returns the one with the highest value.
- UCT method:
    - Calculates the UCT value for a given node.
    - It combines the win rate of the node with an exploration term to balance exploration and exploitation.
- simulate method:
    - Plays out a random simulation from a given node until a terminal state is reached.
    - It selects random moves until the game reaches a terminal state (win, lose, or draw).
- backPropagate method:
    - Updates the node statistics (wins and playouts) back up the tree after a simulation.
    - It increments the playout count and updates the win count based on the result of the simulation.

TicTacToeNode Class

This class represents a node in the search tree specific to the Tic-Tac-Toe game.

- It extends the Node class from the MCTS core and provides methods to interact with the Tic-Tac-Toe game state.

TicTacToe Class

This class represents the Tic-Tac-Toe game itself.

- It defines the game rules, state representation, and methods to interact with the game (e.g., making moves, checking for a winner).

Move and State Classes

These classes define the move and state representations for the Tic-Tac-Toe game.

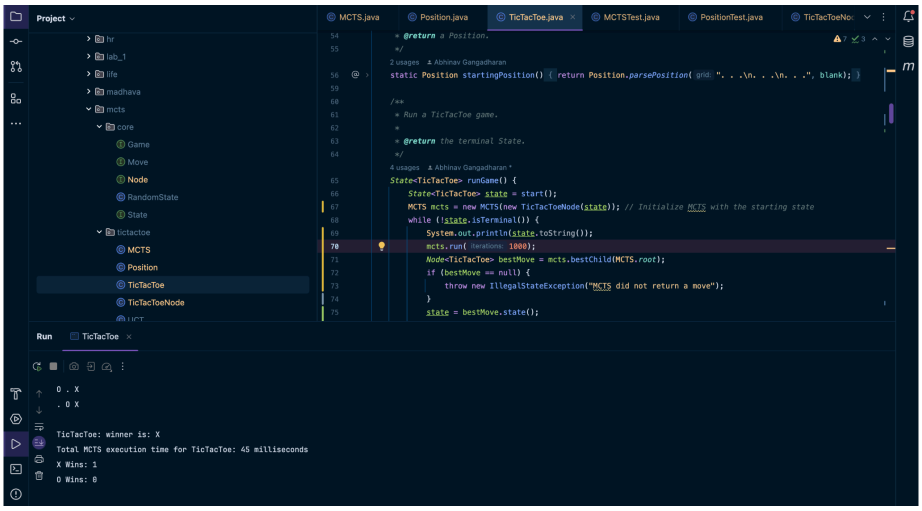
- They provide methods to create and manipulate game moves and states.

TicTacToe Benchmarking:

No. of Games	MCTS Iterations	Time (in Milliseconds)
1	1000	45
1	10000	115
1	100000	555
10	1000	228
10	10000	661
10	100000	4159
100	1000	858
100	10000	5183
100	100000	40509

Timing Screenshots:

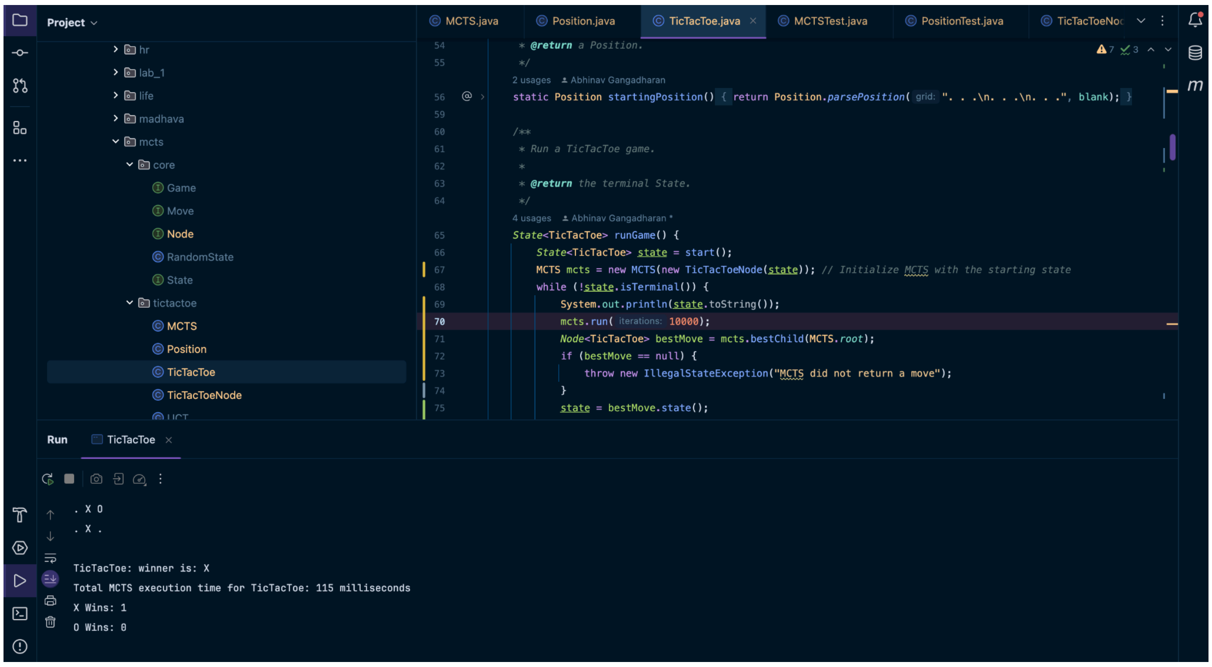
1) For a run with MCTS_Iterations=1000, and only 1 game of Tic-Tac-Toe to be simulated.



```java
     * @return a Position.
     */
    static Position startingPosition() { return Position.parsePosition( grid: ". . .\n. . .\n. . .", blank); }

    /**
     * Run a TicTacToe game.
     *
     * @return the terminal State.
     */
    State<TicTacToe> runGame() {
        State<TicTacToe> state = start();
        MCTS mcts = new MCTS(new TicTacToeNode(state)); // Initialize MCTS with the starting state
        while (!state.isTerminal()) {
            System.out.println(state.toString());
            mcts.run( iterations: 1000);
            Node<TicTacToe> bestMove = mcts.bestChild(MCTS.root);
            if (bestMove == null) {
                throw new IllegalStateException("MCTS did not return a move");
            }
            state = bestMove.state();
```

```
0 . X
. 0 X

TicTacToe: winner is: X
Total MCTS execution time for TicTacToe: 45 milliseconds
X Wins: 1
0 Wins: 0
```

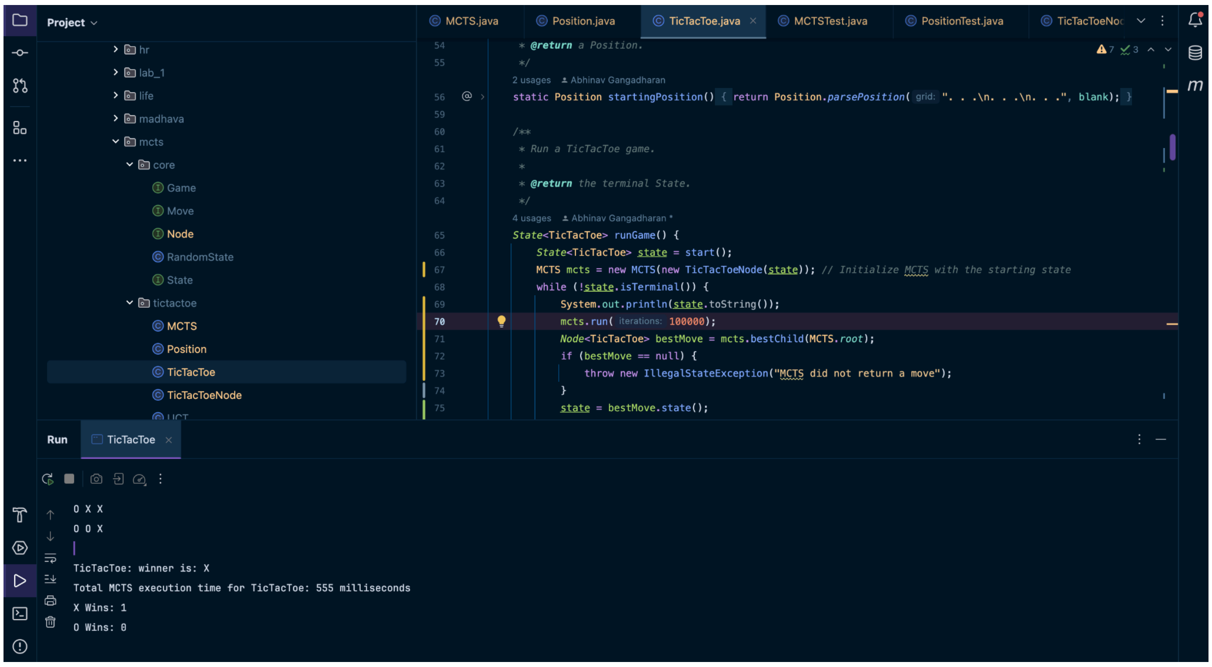
2) For a run with MCTS_Iterations=10000, and only 1 game of Tic-Tac-Toe to be simulated.



```java
     * @return a Position.
     */
    2 usages  Abhinav Gangadharan
    static Position startingPosition() { return Position.parsePosition( ". . .\n. . .\n. . .", blank); }

    /**
     * Run a TicTacToe game.
     *
     * @return the terminal State.
     */
    4 usages  Abhinav Gangadharan *
    State<TicTacToe> runGame() {
        State<TicTacToe> state = start();
        MCTS mcts = new MCTS(new TicTacToeNode(state)); // Initialize MCTS with the starting state
        while (!state.isTerminal()) {
            System.out.println(state.toString());
            mcts.run( 10000);
            Node<TicTacToe> bestMove = mcts.bestChild(MCTS.root);
            if (bestMove == null) {
                throw new IllegalStateException("MCTS did not return a move");
            }
            state = bestMove.state();
```

```
. X 0
. X .

TicTacToe: winner is: X
Total MCTS execution time for TicTacToe: 115 milliseconds
X Wins: 1
0 Wins: 0
```

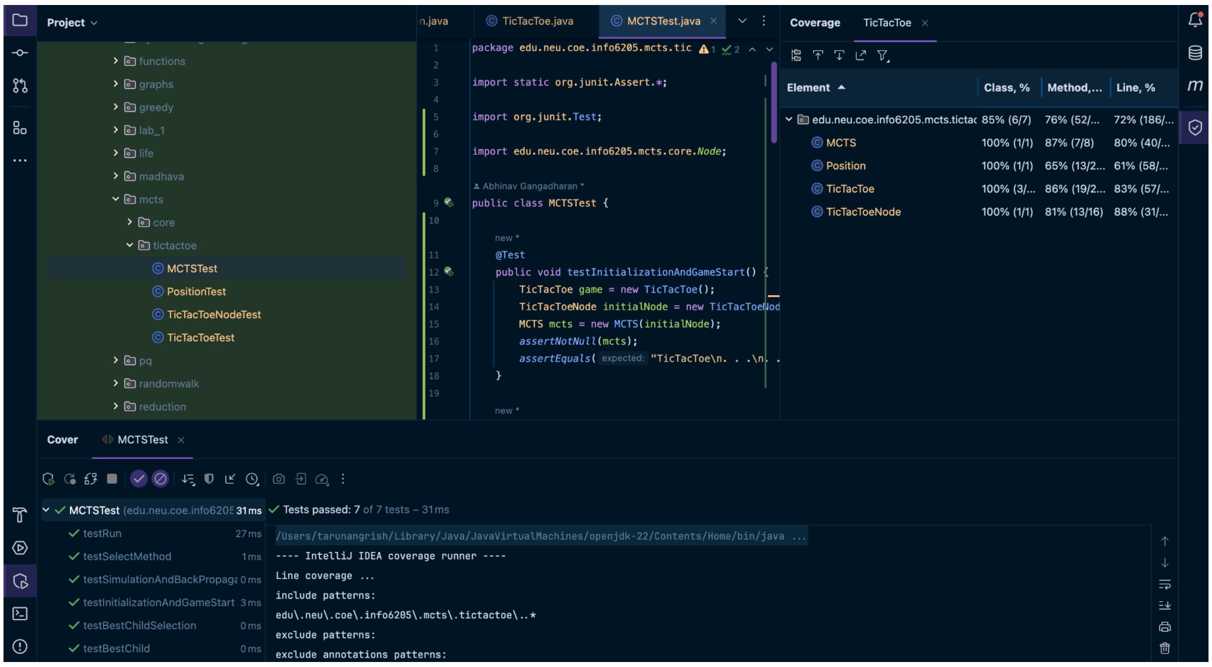
3) For a run with MCTS_Iterations=100000, and only 1 game of Tic-Tac-Toe to be simulated.



```java
     * @return a Position.
     */
    2 usages  Abhinav Gangadharan
    static Position startingPosition() { return Position.parsePosition( ". . .\n. . .\n. . .", blank); }

    /**
     * Run a TicTacToe game.
     *
     * @return the terminal State.
     */
    4 usages  Abhinav Gangadharan *
    State<TicTacToe> runGame() {
        State<TicTacToe> state = start();
        MCTS mcts = new MCTS(new TicTacToeNode(state)); // Initialize MCTS with the starting state
        while (!state.isTerminal()) {
            System.out.println(state.toString());
            mcts.run( 100000);
            Node<TicTacToe> bestMove = mcts.bestChild(MCTS.root);
            if (bestMove == null) {
                throw new IllegalStateException("MCTS did not return a move");
            }
            state = bestMove.state();
```

```
0 X X
0 0 X

TicTacToe: winner is: X
Total MCTS execution time for TicTacToe: 555 milliseconds
X Wins: 1
0 Wins: 0
```

Unit Tests Run with Coverage of all the classes required for successful implementation of Tic-Tac-Toe:

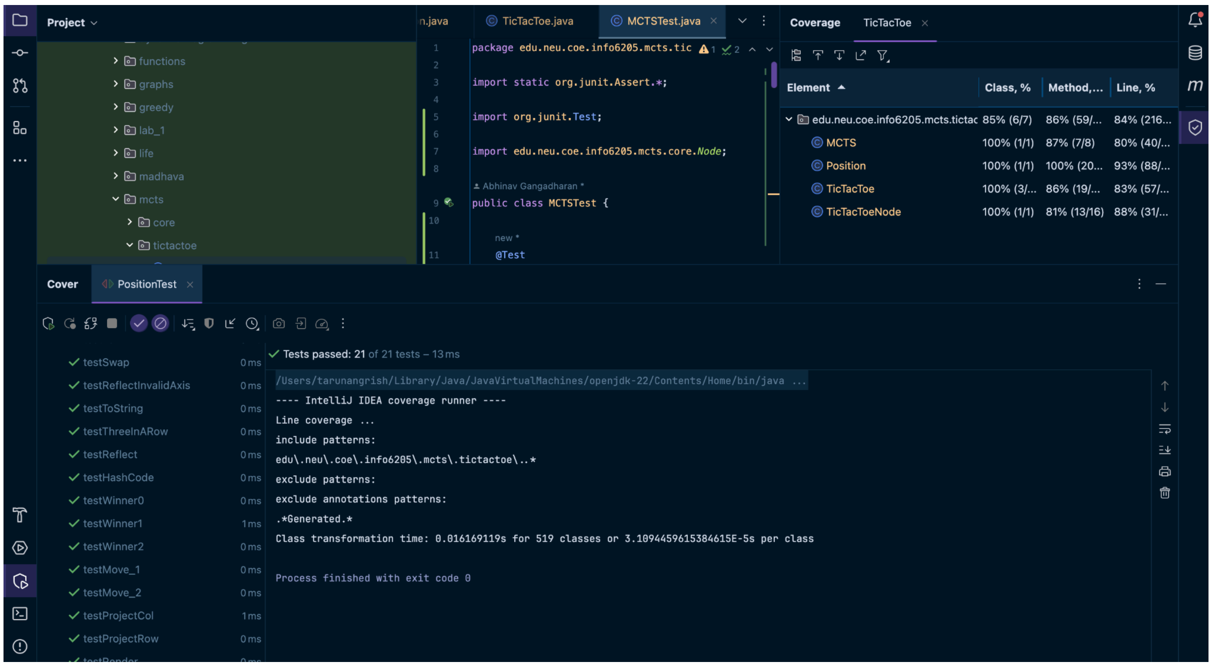


```java
package edu.neu.coe.info6205.mcts.tic

import static org.junit.Assert.*;

import org.junit.Test;

import edu.neu.coe.info6205.mcts.core.Node;

public class MCTSTest {

    @Test
    public void testInitializationAndGameStart() {
        TicTacToe game = new TicTacToe();
        TicTacToeNode initialNode = new TicTacToeNod
        MCTS mcts = new MCTS(initialNode);
        assertNotNull(mcts);
        assertEquals( expected: "TicTacToe\n. . .\n. .
    }
```

Element	Class, %	Method,...	Line, %
edu.neu.coe.info6205.mcts.tictac	85% (6/7)	76% (52/...	72% (186/...
MCTS	100% (1/1)	87% (7/8)	80% (40/...
Position	100% (1/1)	65% (13/2...	61% (58/...
TicTacToe	100% (3/...	86% (19/2...	83% (57/...
TicTacToeNode	100% (1/1)	81% (13/16)	88% (31/...

```
MCTSTest (edu.neu.coe.info6205  31 ms
  testRun                        27 ms
  testSelectMethod                1 ms
  testSimulationAndBackPropaga    0 ms
  testInitializationAndGameStart  3 ms
  testBestChildSelection          0 ms
  testBestChild                   0 ms
```

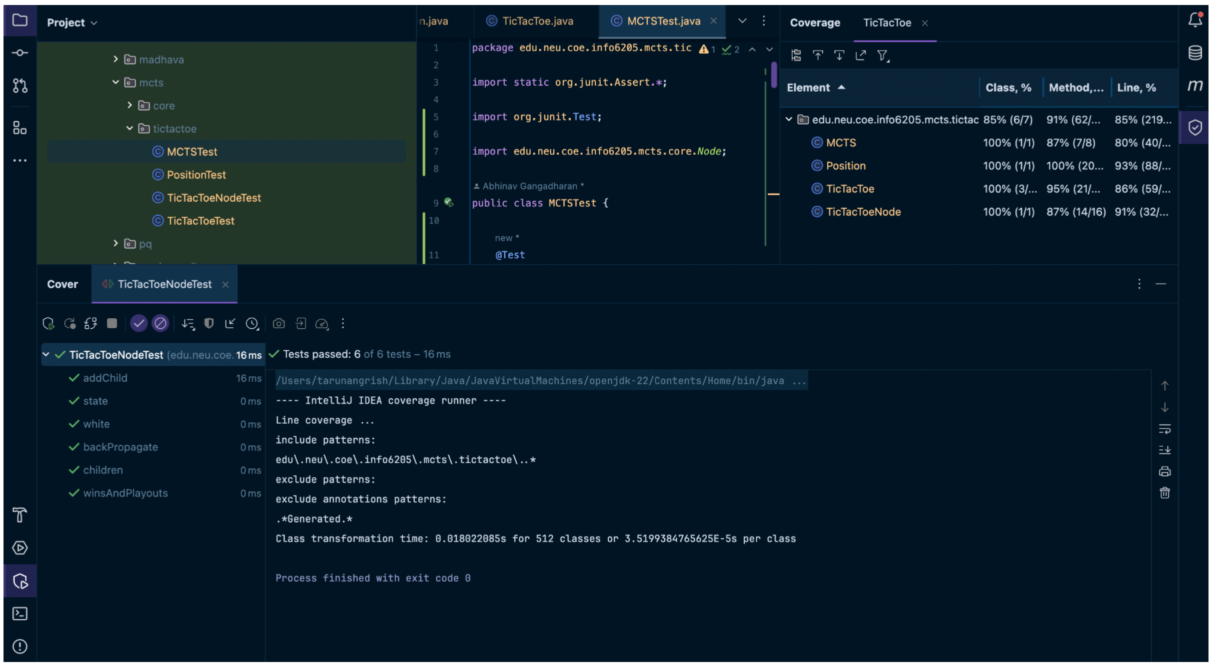
```
Tests passed: 7 of 7 tests – 31 ms
/Users/tarunangrish/Library/Java/JavaVirtualMachines/openjdk-22/Contents/Home/bin/java ...
---- IntelliJ IDEA coverage runner ----
Line coverage ...
include patterns:
edu\.neu\.coe\.info6205\.mcts\.tictactoe\..*
exclude patterns:
exclude annotations patterns:
```



Element	Class, %	Method,...	Line, %
edu.neu.coe.info6205.mcts.tictac	85% (6/7)	86% (59/...	84% (216...
MCTS	100% (1/1)	87% (7/8)	80% (40/...
Position	100% (1/1)	100% (20...	93% (88/...
TicTacToe	100% (3/...	86% (19/...	83% (57/...
TicTacToeNode	100% (1/1)	81% (13/16)	88% (31/...

```
testSwap                 0 ms
testReflectInvalidAxis   0 ms
testToString             0 ms
testThreeInARow          0 ms
testReflect              0 ms
testHashCode             0 ms
testWinner0              0 ms
testWinner1              1 ms
testWinner2              0 ms
testMove_1               0 ms
testMove_2               0 ms
testProjectCol           1 ms
testProjectRow           0 ms
```

```
Tests passed: 21 of 21 tests – 13 ms
/Users/tarunangrish/Library/Java/JavaVirtualMachines/openjdk-22/Contents/Home/bin/java ...
---- IntelliJ IDEA coverage runner ----
Line coverage ...
include patterns:
edu\.neu\.coe\.info6205\.mcts\.tictactoe\..*
exclude patterns:
exclude annotations patterns:
.*Generated.*
Class transformation time: 0.016169119s for 519 classes or 3.1094459615384615E-5s per class

Process finished with exit code 0
```



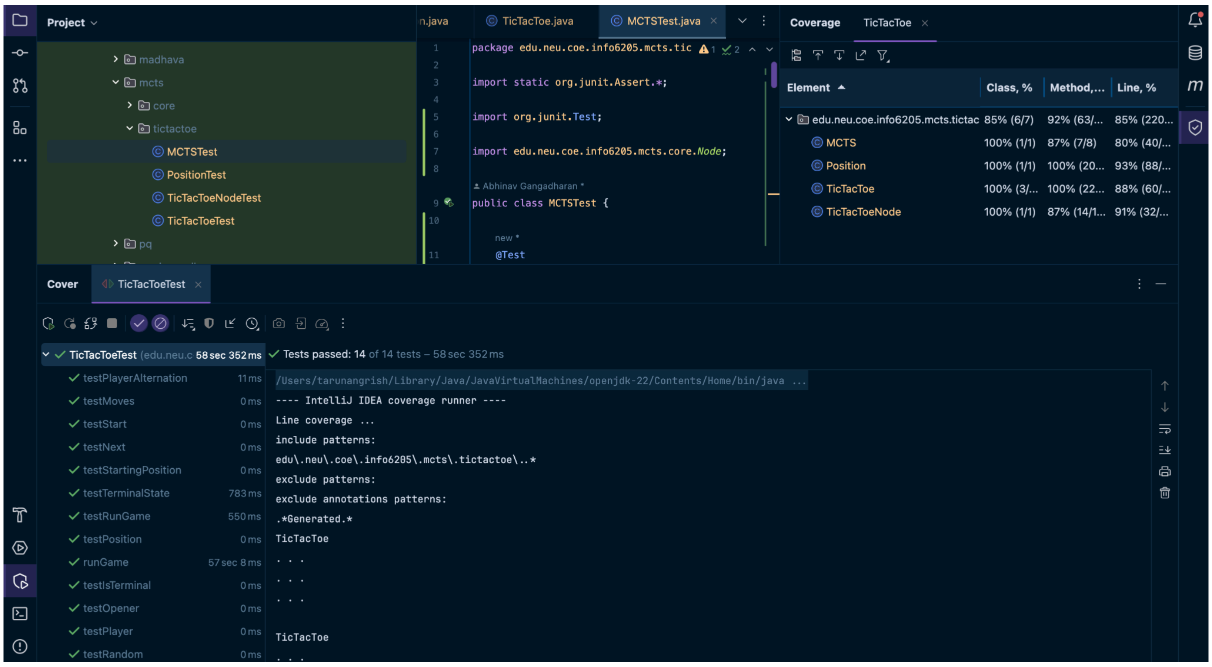
```
package edu.neu.coe.info6205.mcts.tic

import static org.junit.Assert.*;

import org.junit.Test;

import edu.neu.coe.info6205.mcts.core.Node;

Abhinav Gangadharan *
public class MCTSTest {

    new *
    @Test
```

Coverage TicTacToe

Element	Class, %	Method,...	Line, %
edu.neu.coe.info6205.mcts.tictac	85% (6/7)	91% (62/...	85% (219...
MCTS	100% (1/1)	87% (7/8)	80% (40/...
Position	100% (1/1)	100% (20...	93% (88/...
TicTacToe	100% (3/...	95% (21/...	86% (59/...
TicTacToeNode	100% (1/1)	87% (14/16)	91% (32/...

```
TicTacToeNodeTest (edu.neu.coe. 16 ms    Tests passed: 6 of 6 tests – 16 ms
  addChild          16 ms
  state              0 ms
  white              0 ms
  backPropagate      0 ms
  children           0 ms
  winsAndPlayouts    0 ms
```

```
/Users/tarunangrish/Library/Java/JavaVirtualMachines/openjdk-22/Contents/Home/bin/java ...
---- IntelliJ IDEA coverage runner ----
Line coverage ...
include patterns:
edu\.neu\.coe\.info6205\.mcts\.tictactoe\..*
exclude patterns:
exclude annotations patterns:
.*Generated.*
Class transformation time: 0.018022085s for 512 classes or 3.5199384765625E-5s per class

Process finished with exit code 0
```



```
package edu.neu.coe.info6205.mcts.tic

import static org.junit.Assert.*;

import org.junit.Test;

import edu.neu.coe.info6205.mcts.core.Node;

Abhinav Gangadharan *
public class MCTSTest {

    new *
    @Test
```

Coverage TicTacToe

Element	Class, %	Method,...	Line, %
edu.neu.coe.info6205.mcts.tictac	85% (6/7)	92% (63/...	85% (220...
MCTS	100% (1/1)	87% (7/8)	80% (40/...
Position	100% (1/1)	100% (20...	93% (88/...
TicTacToe	100% (3/...	100% (22...	88% (60/...
TicTacToeNode	100% (1/1)	87% (14/1...	91% (32/...

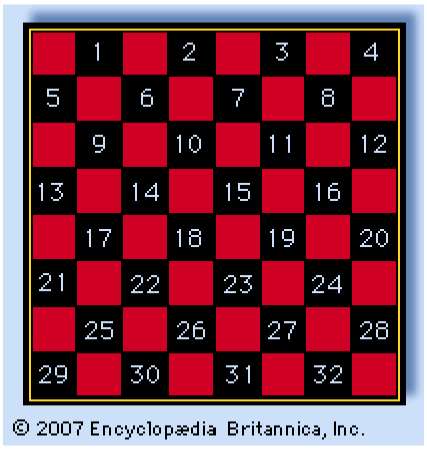
```
TicTacToeTest (edu.neu.c 58 sec 352 ms    Tests passed: 14 of 14 tests – 58 sec 352 ms
  testPlayerAlternation    11 ms
  testMoves                 0 ms
  testStart                 0 ms
  testNext                  0 ms
  testStartingPosition      0 ms
  testTerminalState       783 ms
  testRunGame             550 ms
  testPosition              0 ms
  runGame            57 sec 8 ms
  testIsTerminal            0 ms
  testOpener                0 ms
  testPlayer                0 ms
  testRandom                0 ms
```

```
/Users/tarunangrish/Library/Java/JavaVirtualMachines/openjdk-22/Contents/Home/bin/java ...
---- IntelliJ IDEA coverage runner ----
Line coverage ...
include patterns:
edu\.neu\.coe\.info6205\.mcts\.tictactoe\..*
exclude patterns:
exclude annotations patterns:
.*Generated.*
TicTacToe
. . .
. . .
. . .

TicTacToe
. . .
```

Observations and Results:

During the implementation of MCTS on Tic-Tac-Toe, it was observed that the possibility of winning by a particular player depends on the reward function. If the reward function is skewed in favor of one particular scenario, we can see that the winner of that game would be the same player (irrespective of the fact that who was the opening player). We assumed that the winning player should always be X and skewed the backPropagate method in a way that if the winning_player == X, it should increment the wins by 2 and when we are calculating the Upper Confidence Bound Tree Score, it is bound to return the bestChild of the root(which is the starting point/state of the function) with a higher score, which in our case would be the leafNodes where X is the winning player.

Checkers:
Checkers is a classic board game played on an 8x8 board with 64 squares, typically with pieces in two colors, often red and black. The game is played by two players, each controlling their pieces on opposite sides of the board. The objective of the game is to capture all of your opponent's pieces or block them so they can't make any legal moves.



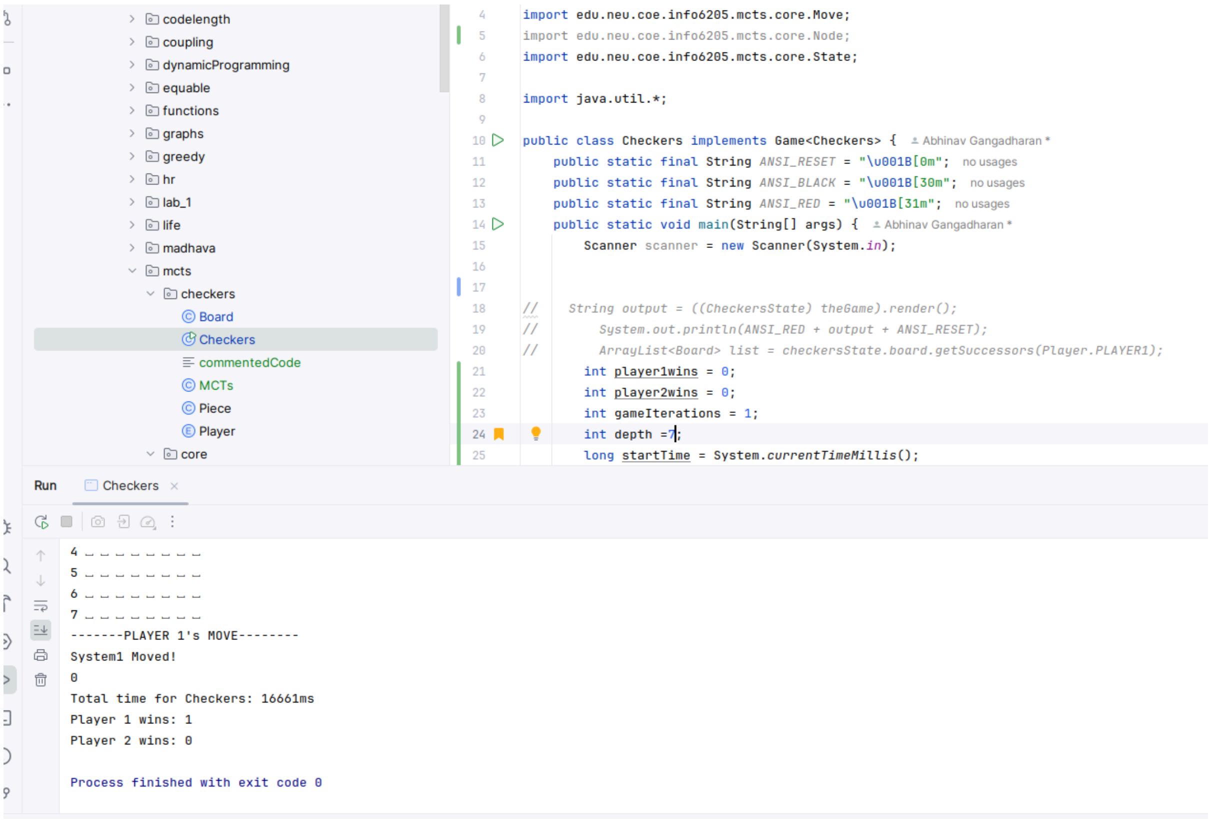
Here are the detailed rules of checkers:

1. Setup: The board is set up so that each player has 12 pieces arranged on the dark squares of the three rows closest to them. The pieces are typically placed on the dark squares, alternating colors, with each player having their pieces on the three rows closest to them.
2. Movement: Pieces can only move diagonally forward, toward the opponent's side of the board. Regular pieces (often called "men") can move one square diagonally forward to an empty square. They capture opponent's pieces by jumping over them diagonally forward into an empty square immediately beyond the captured piece.
3. Kings: When a man reaches the last row on the opponent's side, it becomes a "king." Kings can move and capture both forward and backward diagonally.
4. Jumping: If a player's piece can jump over an opponent's piece, it must do so. Multiple jumps in a single turn are allowed if the opportunity arises. When a piece makes a capture, it must continue jumping as long as possible.
5. King's Jump: If a king can jump over an opponent's piece and continue jumping after the initial jump, it must do so.
6. Ending the Game: The game ends when one player captures all of their opponent's pieces, or when one player is unable to make any legal moves.
7. Stalemate: If neither player can make a legal move, the game ends in a draw or stalemate.
8. Forfeiture: If a player fails to make a required jump, their piece may be captured, and if it results in the loss of the game, the player may forfeit the game.

Checkers Benchmarking:

No. of Games	Depth	Time in Milliseconds
1	4	337
1	5	971
1	6	4325
1	7	16661
3	4	697
3	5	2284
3	6	10550
3	7	48384
8	4	1199
8	5	4118
8	6	24561
8	7	123571

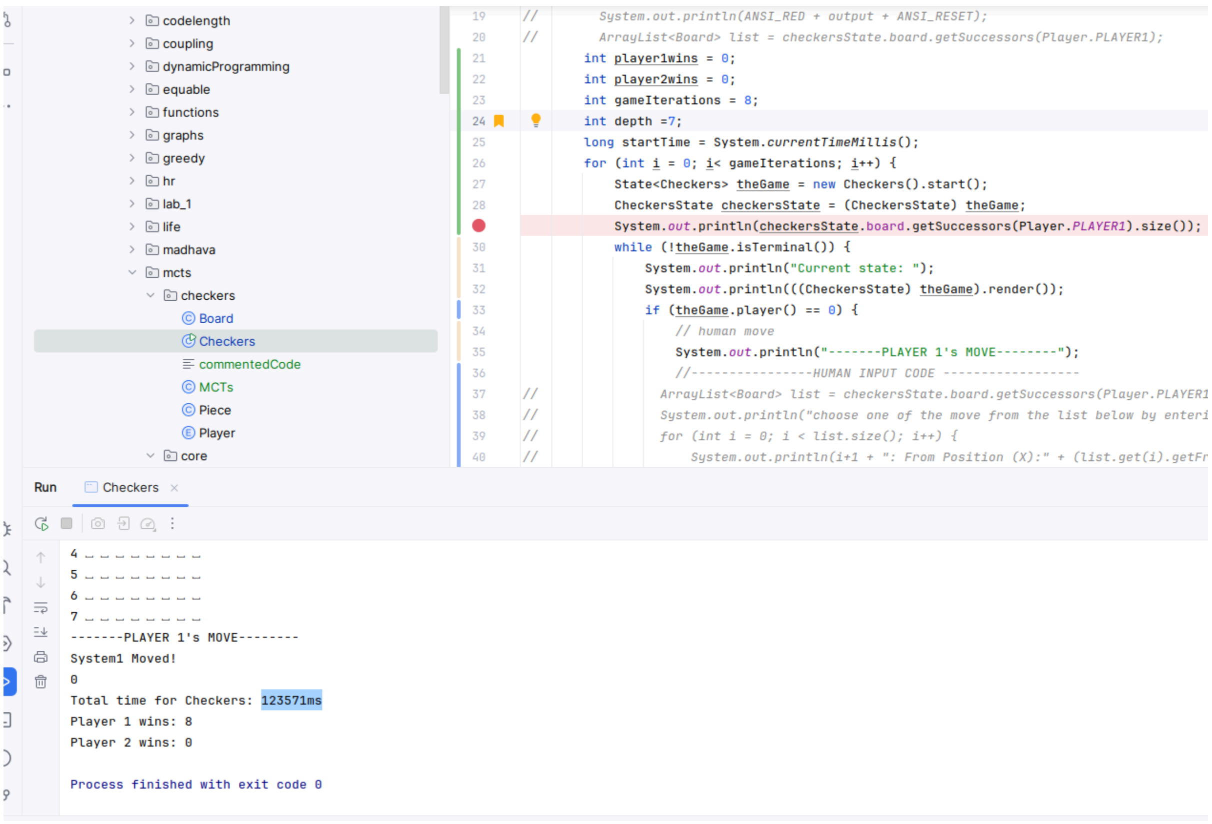
Screenshot for Depth =7 and Iterations = 1



```java
import edu.neu.coe.info6205.mcts.core.Move;
import edu.neu.coe.info6205.mcts.core.Node;
import edu.neu.coe.info6205.mcts.core.State;

import java.util.*;

public class Checkers implements Game<Checkers> {  Abhinav Gangadharan *
    public static final String ANSI_RESET = "\u001B[0m";  no usages
    public static final String ANSI_BLACK = "\u001B[30m";  no usages
    public static final String ANSI_RED = "\u001B[31m";  no usages
    public static void main(String[] args) {  Abhinav Gangadharan *
        Scanner scanner = new Scanner(System.in);


//      String output = ((CheckersState) theGame).render();
//          System.out.println(ANSI_RED + output + ANSI_RESET);
//          ArrayList<Board> list = checkersState.board.getSuccessors(Player.PLAYER1);
        int player1wins = 0;
        int player2wins = 0;
        int gameIterations = 1;
        int depth =7;
        long startTime = System.currentTimeMillis();
```

```
4 _ _ _ _ _ _ _ _
5 _ _ _ _ _ _ _ _
6 _ _ _ _ _ _ _ _
7 _ _ _ _ _ _ _ _
-------PLAYER 1's MOVE--------
System1 Moved!
0
Total time for Checkers: 16661ms
Player 1 wins: 1
Player 2 wins: 0

Process finished with exit code 0
```

Screenshot for Depth =7 and Iterations = 8



```java
//          System.out.println(ANSI_RED + output + ANSI_RESET);
//          ArrayList<Board> list = checkersState.board.getSuccessors(Player.PLAYER1);
        int player1wins = 0;
        int player2wins = 0;
        int gameIterations = 8;
        int depth =7;
        long startTime = System.currentTimeMillis();
        for (int i = 0; i< gameIterations; i++) {
            State<Checkers> theGame = new Checkers().start();
            CheckersState checkersState = (CheckersState) theGame;
            System.out.println(checkersState.board.getSuccessors(Player.PLAYER1).size());
            while (!theGame.isTerminal()) {
                System.out.println("Current state: ");
                System.out.println(((CheckersState) theGame).render());
                if (theGame.player() == 0) {
                    // human move
                    System.out.println("-------PLAYER 1's MOVE--------");
                    //---------------HUMAN INPUT CODE -----------------
//                  ArrayList<Board> list = checkersState.board.getSuccessors(Player.PLAYER1
//                  System.out.println("choose one of the move from the list below by enteri
//                  for (int i = 0; i < list.size(); i++) {
//                      System.out.println(i+1 + ": From Position (X):" + (list.get(i).getFr
```

```
4 _ _ _ _ _ _ _ _
5 _ _ _ _ _ _ _ _
6 _ _ _ _ _ _ _ _
7 _ _ _ _ _ _ _ _
-------PLAYER 1's MOVE--------
System1 Moved!
0
Total time for Checkers: 123571ms
Player 1 wins: 8
Player 2 wins: 0

Process finished with exit code 0
```

Initial Methodology:
We had initially implemented code which attempted to explore all the nodes upto terminal condition, but were (as expected) met with errors as Java would give up after a while.

Improvements:
As a workaround to the issue we faced before, we decided to implement a limit to how deep a node would expand. With this, we were able to execute the program.

Observations:
Whichever player started the game was coming up with a higher probability of winning.

References:
https://int8.io/monte-carlo-tree-search-beginners-guide/
https://www.baeldung.com/java-monte-carlo-tree-search
https://www.researchgate.net/figure/Game-Tree-for-Tic-Tac-Toe_fig2_290786914