INFO 6205 - Program Structures and Algorithms

Spring 2025

Team Members:

Harshith Patil - **002374708**,

Omisha Kataria - **002813839,**

Prajeshkumar Sundareswaran - **002474063**,

GitHub link : https://github.com/Omisha-Kataria/INFO-6205_Project

Gomoku:

Gomoku, also known as Five in a Row, is a traditional two-player strategy board game played on a 15×15 grid where players take turns placing stones of their color with the goal of forming an unbroken line of five stones horizontally, vertically, or diagonally before their opponent. With its elegantly simple rules yet complex strategic depth, Gomoku provides an excellent environment for testing artificial intelligence algorithms. Despite its straightforward objective, Gomoku has an immense game tree complexity, with approximately 10^170 possible board positions in the standard 15×15 grid, making it significantly more complex than checkers (10^20) and closer to chess (10^120) in terms of possible game states. This combination of simple rules and deep complexity makes Gomoku an ideal candidate for exploring the effectiveness of Monte Carlo Tree Search algorithms.

Monte Carlo Tree Search (MCTS):

Monte Carlo Tree Search is a decision-making algorithm used in games like Gomoku. It works by repeatedly simulating potential game outcomes to identify strong moves. The algorithm balances trying new possibilities (exploration) with focusing on moves that have worked well previously (exploitation). MCTS has four simple steps that repeat many times: select promising moves, expand the search tree with new positions, play random moves until the game ends, and update statistics based on the results. Unlike other game algorithms, MCTS doesn't need complex game-specific evaluation functions, making it versatile for different games. The more iterations it performs, the stronger it becomes, allowing it to find good moves even in complex games with many possible positions.

TicTacToe implementation:

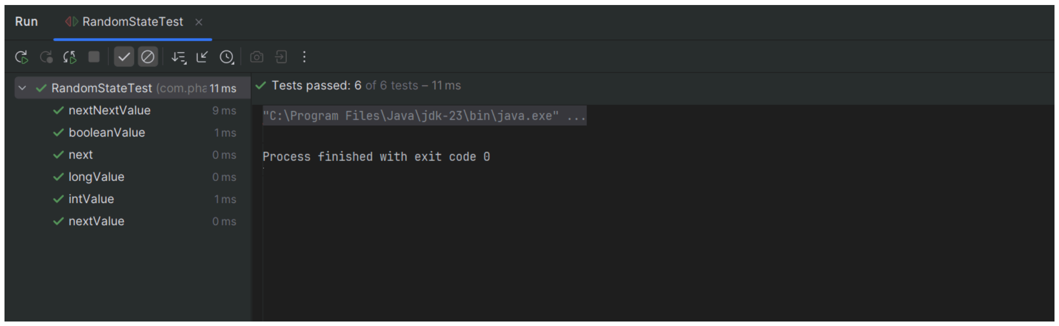
<u>Design:</u>

- The core abstractions live in com.phasmidsoftware.dsaipg.projects.mcts.core (Game, State, Move, Node, RandomState).
- TicTacToe and its inner TicTacToeState encode the rules, deliver legal moves, detect wins/draws and hold a Position object (a 3 × 3 int grid: 1 = X, 0 = O, −1 = empty).
- TicTacToeNode is a concrete Node<TicTacToe> implementation. Each node stores a parent pointer, the move that produced it, a mutable list of children, and two statistics (wins, playouts). Helper methods include expandAll(), incrementPlayouts(), addWins() and getParent().
- MCTS performs the canonical Monte‑Carlo Tree Search loop — select → expand → simulate → back‑propagate — using UCB1/UCT for the tree policy. After a fixed budget of playouts (**5000 by default**) the driver chooses the child with the highest empirical win‑rate and returns a fresh node rooted at that state. Each of the four phases is timed with System.nanoTime; accumulators are public for test inspection.

<u>Algorithms:</u>

- Selection walks the tree with UCT. The first unvisited child is always expanded if one exists. Expansion is breadth‑first once per step (all children are created by expandAll()), then a random child is chosen for rollout.
- **Simulation / rollout** plays purely random moves until terminal. Rewards are recorded as 2 for a win, 1 for a draw, 0 for a loss from the perspective of the playout's starting player. Back‑propagation walks parent links, incrementing playouts and adding either the reward or 2 − reward depending on whose point of view the node represents.
- Action selection after the playout budget returns the child with the highest wins / playouts ratio. Switching back to the classical "**most‑visited**" policy is a one‑line change in nextNodeWithTiming.

Test analysis:



Run | RandomStateTest

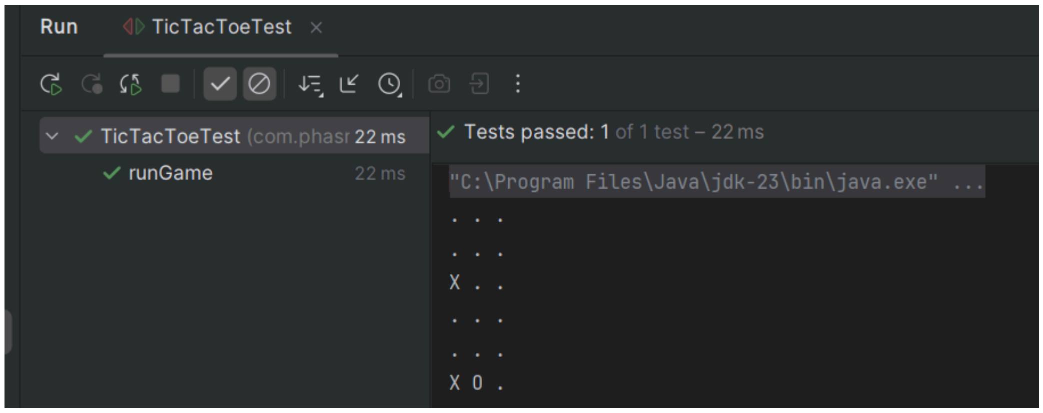
RandomStateTest (com.pha 11 ms
- nextNextValue 9 ms
- booleanValue 1 ms
- next 0 ms
- longValue 0 ms
- intValue 1 ms
- nextValue 0 ms

Tests passed: 6 of 6 tests – 11 ms

```
"C:\Program Files\Java\jdk-23\bin\java.exe" ...

Process finished with exit code 0
```

RandomStateTest



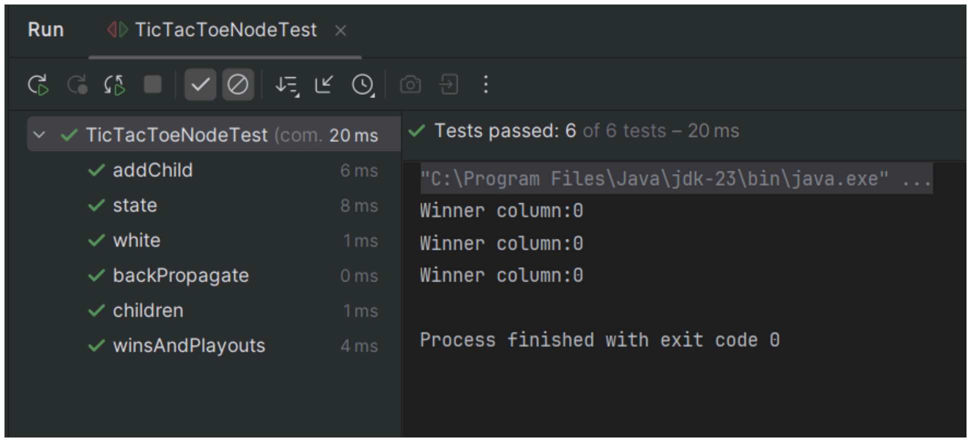
Run | TicTacToeTest

TicTacToeTest (com.phasr 22 ms
- runGame 22 ms

Tests passed: 1 of 1 test – 22 ms

```
"C:\Program Files\Java\jdk-23\bin\java.exe" ...
. . .
. . .
X . .
. . .
. . .
X O .
```

TicTacToeTest



Run | TicTacToeNodeTest

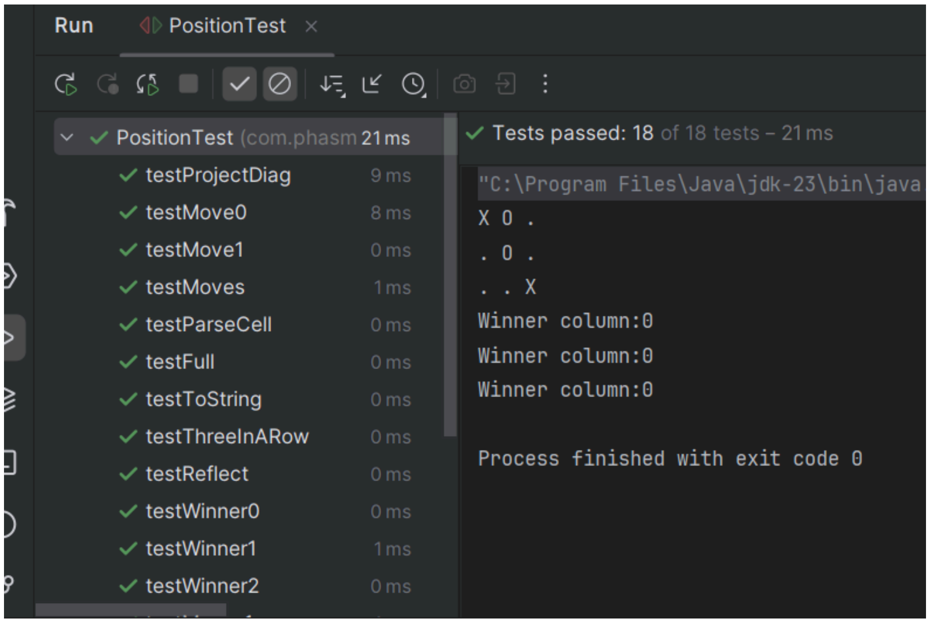
TicTacToeNodeTest (com. 20 ms
- addChild 6 ms
- state 8 ms
- white 1 ms
- backPropagate 0 ms
- children 1 ms
- winsAndPlayouts 4 ms

Tests passed: 6 of 6 tests – 20 ms

```
"C:\Program Files\Java\jdk-23\bin\java.exe" ...
Winner column:0
Winner column:0
Winner column:0

Process finished with exit code 0
```

TicTacToeNodeTest

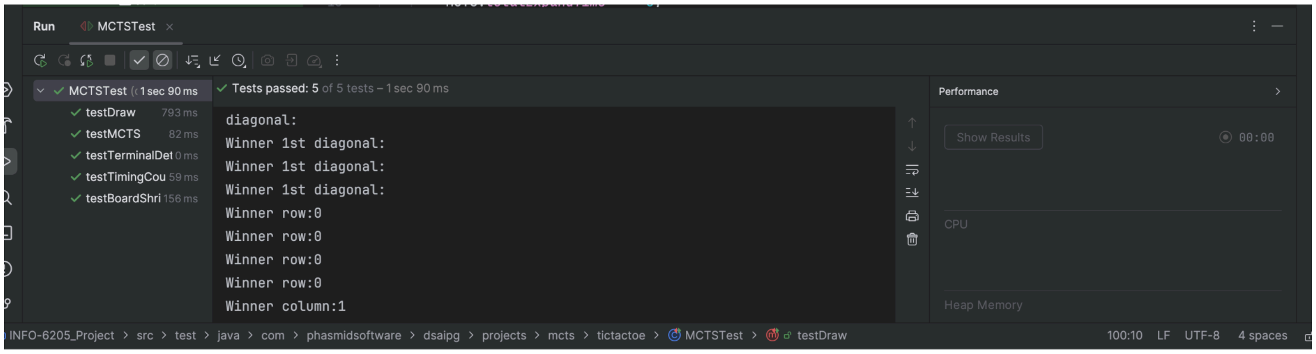


```
Run   PositionTest ×

PositionTest (com.phasm 21 ms      ✓ Tests passed: 18 of 18 tests – 21 ms
  ✓ testProjectDiag   9 ms          "C:\Program Files\Java\jdk-23\bin\java
  ✓ testMove0         8 ms          X O .
  ✓ testMove1         0 ms          . O .
  ✓ testMoves         1 ms          . . X
  ✓ testParseCell     0 ms          Winner column:0
  ✓ testFull          0 ms          Winner column:0
  ✓ testToString      0 ms          Winner column:0
  ✓ testThreeInARow   0 ms
  ✓ testReflect       0 ms          Process finished with exit code 0
  ✓ testWinner0       0 ms
  ✓ testWinner1       1 ms
  ✓ testWinner2       0 ms
```

PositionTest

MCTS TicTacToe test



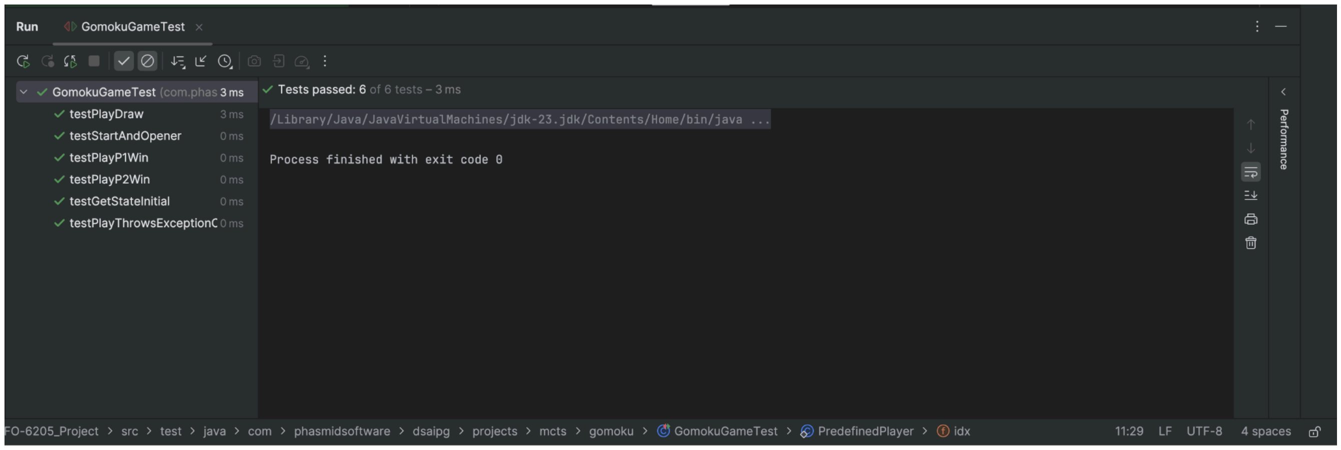
```
Run   MCTSTest ×

MCTSTest (c 1 sec 90 ms            ✓ Tests passed: 5 of 5 tests – 1 sec 90 ms      Performance
  ✓ testDraw         793 ms         diagonal:
  ✓ testMCTS          82 ms         Winner 1st diagonal:                         Show Results    00:00
  ✓ testTerminalDet    0 ms         Winner 1st diagonal:
  ✓ testTimingCou     59 ms         Winner 1st diagonal:
  ✓ testBoardShri    156 ms         Winner row:0                                 CPU
                                    Winner row:0
                                    Winner row:0
                                    Winner row:0
                                    Winner column:1                              Heap Memory

INFO-6205_Project > src > test > java > com > phasmidsoftware > dsaipg > projects > mcts > tictactoe > MCTSTest > testDraw     100:10  LF  UTF-8  4 spaces
```

GomokuTests

GomokuGameTest



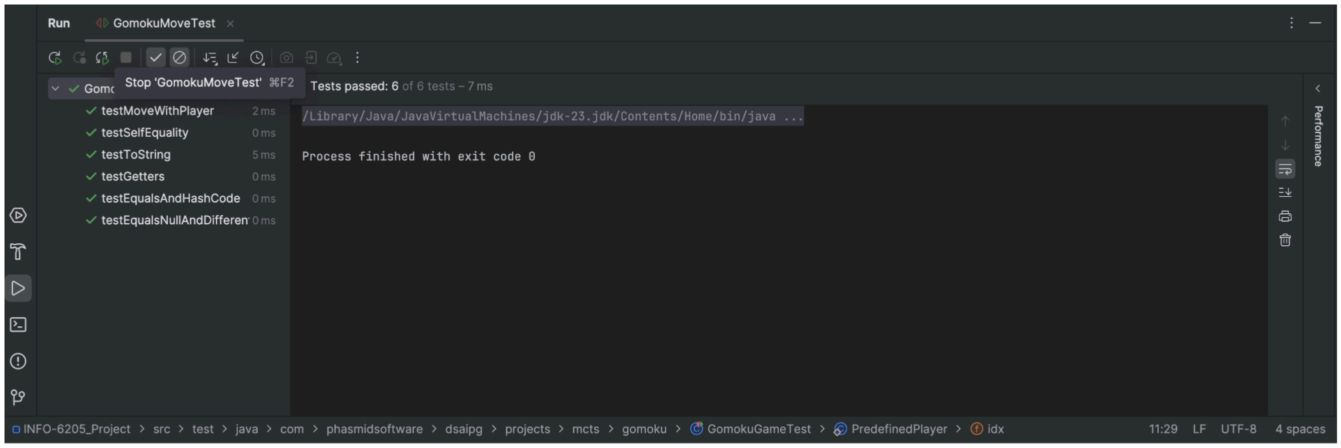
```
GomokuGameTest (com.phas 3 ms
  testPlayDraw 3 ms
  testStartAndOpener 0 ms
  testPlayP1Win 0 ms
  testPlayP2Win 0 ms
  testGetStateInitial 0 ms
  testPlayThrowsExceptionC 0 ms
```

```
Tests passed: 6 of 6 tests – 3 ms
/Library/Java/JavaVirtualMachines/jdk-23.jdk/Contents/Home/bin/java ...

Process finished with exit code 0
```

FO-6205_Project > src > test > java > com > phasmidsoftware > dsaipg > projects > mcts > gomoku > GomokuGameTest > PredefinedPlayer > idx 11:29 LF UTF-8 4 spaces

GomokuMoveTest



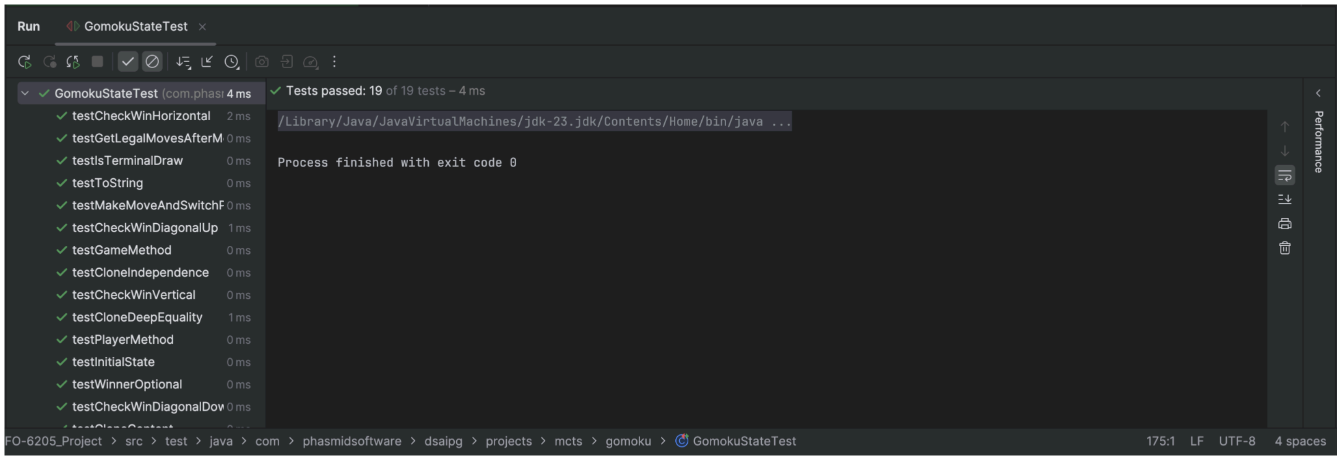
```
Gomc Stop 'GomokuMoveTest' ⌘F2
  testMoveWithPlayer 2 ms
  testSelfEquality 0 ms
  testToString 5 ms
  testGetters 0 ms
  testEqualsAndHashCode 0 ms
  testEqualsNullAndDifferen 0 ms
```

```
Tests passed: 6 of 6 tests – 7 ms
/Library/Java/JavaVirtualMachines/jdk-23.jdk/Contents/Home/bin/java ...

Process finished with exit code 0
```

INFO-6205_Project > src > test > java > com > phasmidsoftware > dsaipg > projects > mcts > gomoku > GomokuGameTest > PredefinedPlayer > idx 11:29 LF UTF-8 4 spaces

GomokuStateTest



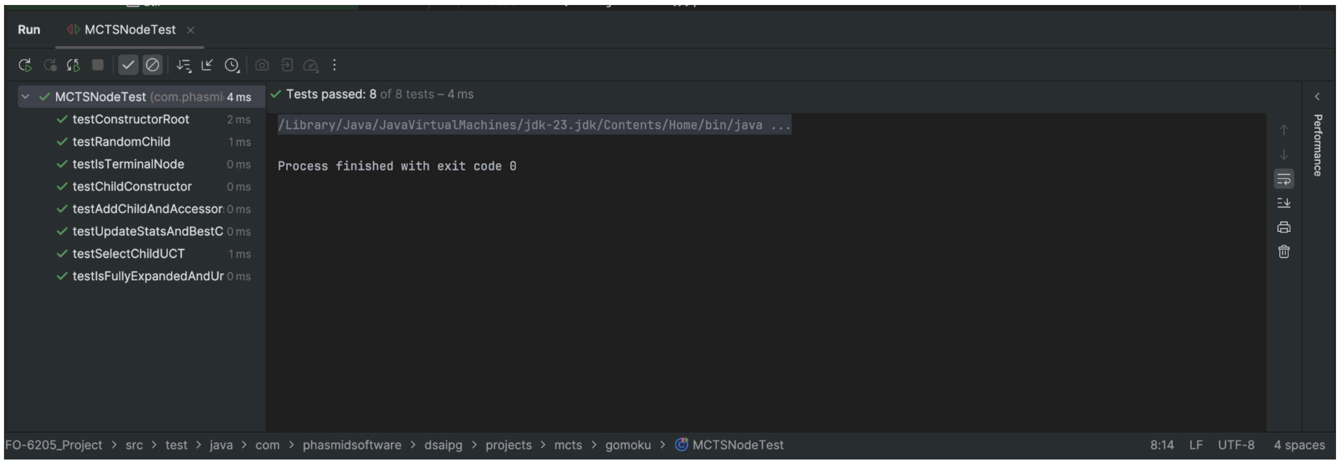
```
GomokuStateTest (com.phasi 4 ms
  testCheckWinHorizontal 2 ms
  testGetLegalMovesAfterM 0 ms
  testIsTerminalDraw 0 ms
  testToString 0 ms
  testMakeMoveAndSwitchF 0 ms
  testCheckWinDiagonalUp 1 ms
  testGameMethod 0 ms
  testCloneIndependence 0 ms
  testCheckWinVertical 0 ms
  testCloneDeepEquality 1 ms
  testPlayerMethod 0 ms
  testInitialState 0 ms
  testWinnerOptional 0 ms
  testCheckWinDiagonalDov 0 ms
```

```
Tests passed: 19 of 19 tests – 4 ms
/Library/Java/JavaVirtualMachines/jdk-23.jdk/Contents/Home/bin/java ...

Process finished with exit code 0
```

FO-6205_Project > src > test > java > com > phasmidsoftware > dsaipg > projects > mcts > gomoku > GomokuStateTest 175:1 LF UTF-8 4 spaces

MCTSNodeTest


Run MCTSNodeTest

MCTSNodeTest (com.phasmi 4 ms
testConstructorRoot 2 ms
testRandomChild 1 ms
testIsTerminalNode 0 ms
testChildConstructor 0 ms
testAddChildAndAccessor 0 ms
testUpdateStatsAndBestC 0 ms
testSelectChildUCT 1 ms
testIsFullyExpandedAndUr 0 ms

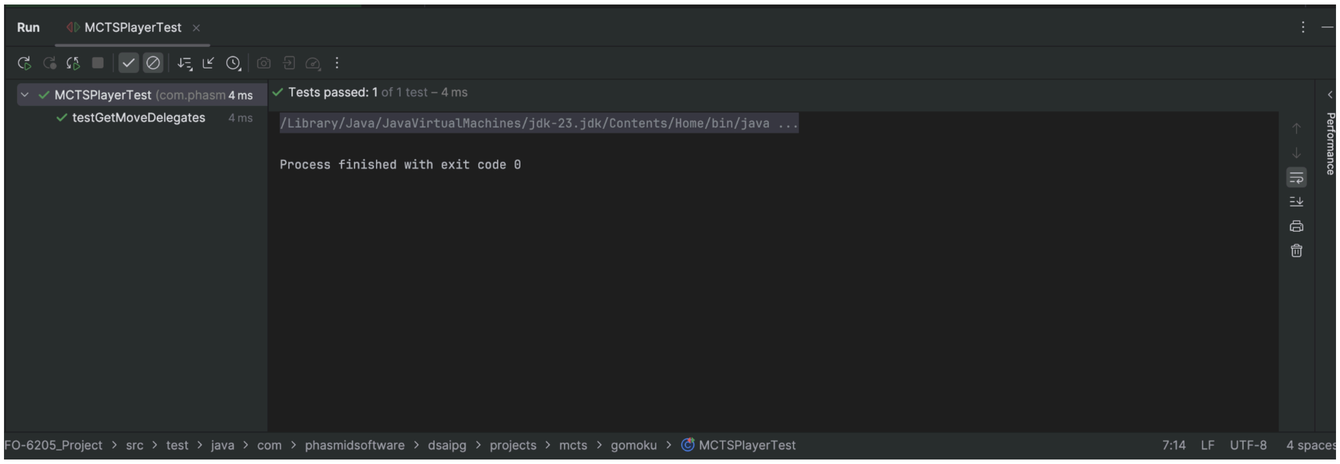
Tests passed: 8 of 8 tests – 4 ms

```
/Library/Java/JavaVirtualMachines/jdk-23.jdk/Contents/Home/bin/java ...

Process finished with exit code 0
```

FO-6205_Project > src > test > java > com > phasmidsoftware > dsaipg > projects > mcts > gomoku > MCTSNodeTest 8:14 LF UTF-8 4 spaces

MCTSPlayerTest


Run MCTSPlayerTest

MCTSPlayerTest (com.phasm 4 ms
testGetMoveDelegates 4 ms

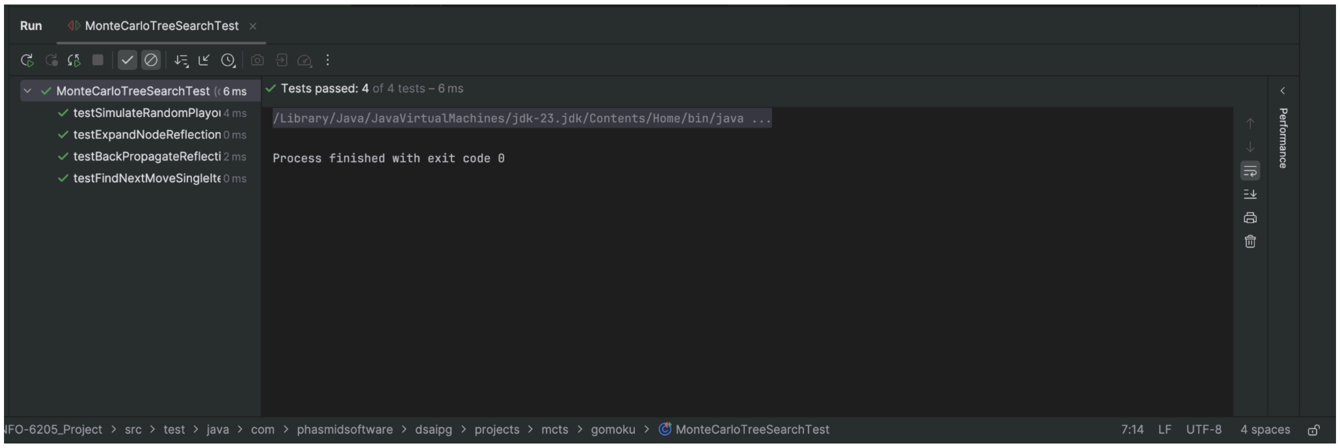
Tests passed: 1 of 1 test – 4 ms

```
/Library/Java/JavaVirtualMachines/jdk-23.jdk/Contents/Home/bin/java ...

Process finished with exit code 0
```

FO-6205_Project > src > test > java > com > phasmidsoftware > dsaipg > projects > mcts > gomoku > MCTSPlayerTest 7:14 LF UTF-8 4 spaces

MonteCarloTreeSearchTest


Run MonteCarloTreeSearchTest

MonteCarloTreeSearchTest (c 6 ms
testSimulateRandomPlayo 4 ms
testExpandNodeReflection 0 ms
testBackPropagateReflecti 2 ms
testFindNextMoveSingleIte 0 ms

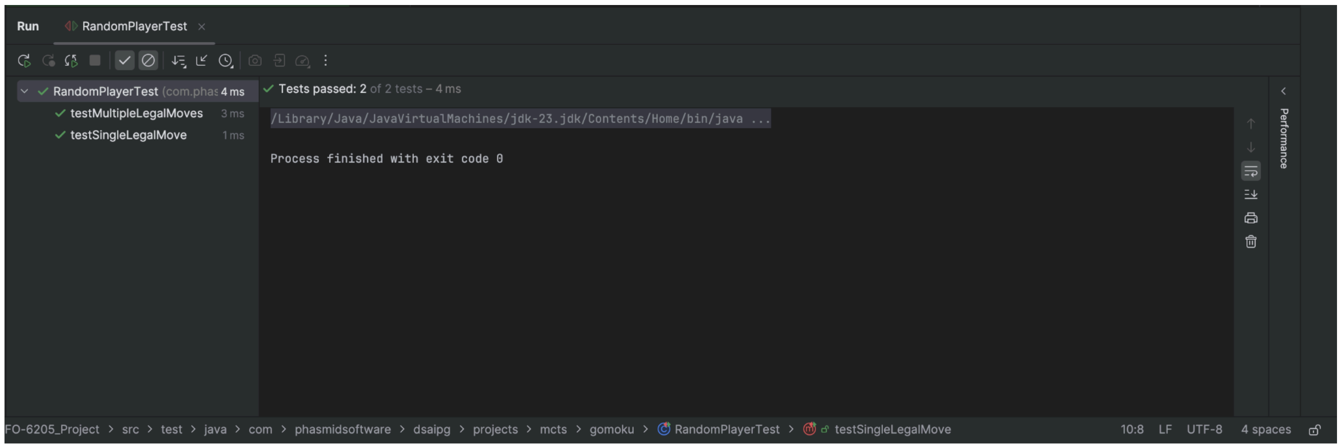
Tests passed: 4 of 4 tests – 6 ms

```
/Library/Java/JavaVirtualMachines/jdk-23.jdk/Contents/Home/bin/java ...

Process finished with exit code 0
```

NFO-6205_Project > src > test > java > com > phasmidsoftware > dsaipg > projects > mcts > gomoku > MonteCarloTreeSearchTest 7:14 LF UTF-8 4 spaces

RandomPlayerTest


Run
RandomPlayerTest
RandomPlayerTest (com.phas 4 ms
testMultipleLegalMoves 3 ms
testSingleLegalMove 1 ms
Tests passed: 2 of 2 tests – 4 ms
Performance
FO-6205_Project > src > test > java > com > phasmidsoftware > dsaipg > projects > mcts > gomoku > RandomPlayerTest > testSingleLegalMove
10:8 LF UTF-8 4 spaces

```
/Library/Java/JavaVirtualMachines/jdk-23.jdk/Contents/Home/bin/java ...

Process finished with exit code 0
```

<u>Observations on tests and experiments:</u>

- With 5000 playouts per move a full game finishes in well under a second on a modern laptop. Memory usage is **minimal** because the tree is at most nine plies deep and each node stores only primitive fields and a small child list**.**

- The code was structured clearly by separating the game rules, the node structure (used in the search tree), and the MCTS logic. This made the code easier to understand, maintain, and test.

- Detailed timing information was added for each phase of the MCTS process (selection, expansion, simulation, and backpropagation). This helped us analyze and understand how the algorithm behaves during execution.

- A full set of test cases was written and passed successfully, which ensures that the algorithm works correctly in different scenarios like draws, wins, and forced moves.

- The design allows for easy changes in the future—for example, testing different move strategies (rollout policies) or using a bigger board like **15x15 for Gomoku.**

<u>Conclusion:</u>

The updated project keeps the game rules, node structure, and search steps organized in separate sections. It tracks how long each part takes, passes all the tests, and is easy to modify if you want to try new strategies or use a bigger board.

GOMOKU Implementation:

Board & Move Representation

- GomokuState holds a boardSize×boardSize int[][], currentPlayer, and provides getLegalMoves(), makeMove(GomokuMove), isTerminal(), plus checkWin() (horizontal, vertical, both diagonals).

- GomokuMove is a simple (row, col) pair (implements Move<GomokuGame>).

Game Wrapper

- GomokuGame implements your core Game<GomokuGame> interface:

- start() returns a fresh GomokuState.

- opener() = **PLAYER_ONE**.

- Owns two Player objects and a loop in play() that calls **getMove(...)** until **isTerminal()**.

Tree Node for MCTS

- **MCTSNode** wraps a GomokuState, parent/children pointers, statistics (wins, visits), and the move that led here.

- Methods for expansion (getUntriedMoves(), addChild(...)), selection (selectChild() via UCT), and backpropagation (updateStats(...), bestChild(...)).

Monte Carlo Tree Search

- MonteCarloTreeSearch (with an iterationLimit):

    1. Selection: descend via UCT until a leaf.

    2. Expansion: if non‑terminal, expand all its untried moves.

    3. Simulation: from one child (or leaf) play random moves to the end.

    4. Backpropagation: propagate win/draw results up the path.

- **findNextMove(rootState)** returns the child of the root with **highest visits**.

Players

- **MCTSPlayer** wraps a MonteCarloTreeSearch and calls **findNextMove(...)** each turn.
- **RandomPlayer** picks uniformly from **state.getLegalMoves()**.

UI & Experimentation

GomokuUI: Swing‑based board display, human vs. AI interaction, restart button, status label.

GomokuExperiment: main(...) that benchmarks "MCTS vs Random" and "MCTS vs MCTS" across varied iteration‑counts and game‑counts, reporting avg ms and win/draw stats.

With these six components in place and wired together, you have a fully playable (and testable!) Gomoku implementation powered by MCTS.

Gomoku results:

We conducted experiments with N games of Gomoku played between two types of agents: Player 1 using Monte Carlo Tree Search (MCTS) and Player 2 using either a random move strategy or another MCTS agent. Similar to Connect Four, Gomoku is a relatively complex game with longer gameplay, and thus, we kept the units of runtime in milliseconds to reflect the computational scale.

The objective of these experiments was to analyze the performance of the MCTS algorithm across different iteration values and observe its behavior both against random strategies and other MCTS players.

Project Demo Video

Please find the demo video in the repository

Test Results:

MCTS vs Random results:

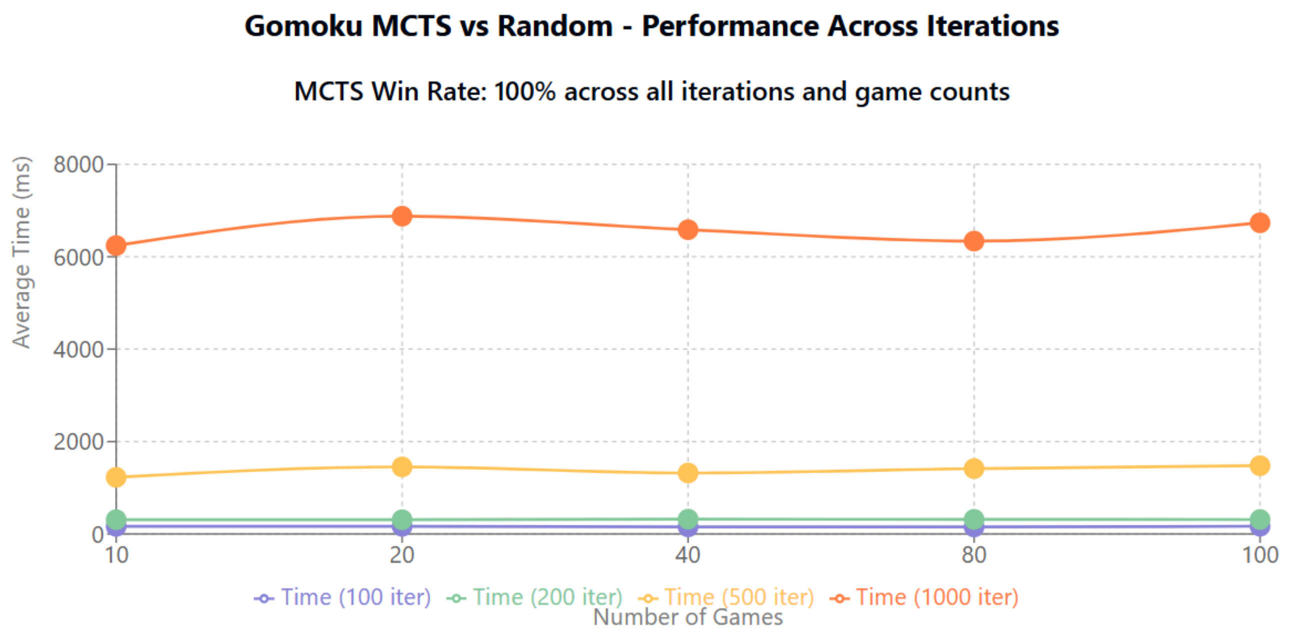
Gomoku MCTS vs Random (iters = 100)					
Games	**10**	**20**	**40**	**80**	**100**
Player 1 (MCTS)	10	20	40	80	100
Player 2 (Random)	0	0	0	0	0
Draws	0	0	0	0	0

Avg Time	164.7	165.1	153.475	151.1	167.81

Gomoku MCTS vs Random (iters = 200)					
Games	**10**	**20**	**40**	**80**	**100**
Player 1 (MCTS)	10	20	40	80	100
Player 2 (Random)	0	0	0	0	0
Draws	0	0	0	0	0
Avg Time	308.4	307.05	320.125	315.463	312.86

Gomoku MCTS vs Random (iters = 500)					
Games	**10**	**20**	**40**	**80**	**100**
Player 1 (MCTS)	10	20	40	80	100
Player 2 (Random)	0	0	0	0	0
Draws	0	0	0	0	0
Avg Time	1225.4	1451.05	1319.5	1414.588	1477.2

Gomoku MCTS vs Random (iters = 1000)					
Games	**10**	**20**	**40**	**80**	**100**
Player 1 (MCTS) wins	10	20	40	80	100
Player 2 (Random) wins	0	0	0	0	0
Draws	0	0	0	0	0
Avg Time	6241.4	6877.55	6584	6338.2	6733.86



Key Observations:

- MCTS consistently achieves 100% win rate against Random player regardless of iteration count
- Average computation time scales approximately linearly with iteration count
- Computation time remains fairly consistent across different game counts for the same iteration setting
- 1000 iterations takes approximately 4-5 times longer than 500 iterations

MCTS vs MCTS results:

Gomoku MCTS vs MCTS (iters = 100)					
Games	**10**	**20**	**40**	**80**	**100**
Player 1 wins	10	20	40	80	100
Player 2 wins	0	0	0	0	0
Draws	0	0	0	0	0
Avg Time	1381.5	1375.7	1372.075	1395.288	1412.25

Gomoku MCTS vs MCTS (iters = 200)					
Games	**10**	**20**	**40**	**80**	**100**
Player 1 wins	6	12	18	39	57
Player 2 wins	4	8	22	41	43
Draws	0	0	0	0	0

Avg Time	3037.5	3001.95	2997.225	2992.425	2977.41

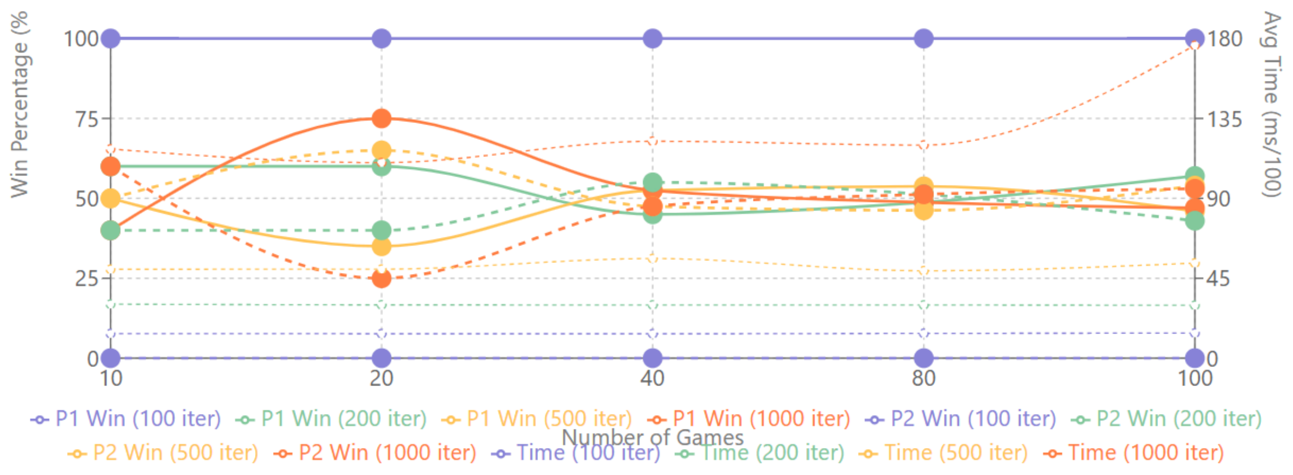
Gomoku MCTS vs MCTS (iters = 500)					
Games	**10**	**20**	**40**	**80**	**100**
Player 1 wins	5	7	21	43	46
Player 2 wins	5	13	19	37	54
Draws	0	0	0	0	0
Avg Time	5003.8	5010.6	5606.8	4922.638	5354.37

Gomoku MCTS vs MCTS (iters = 1000)					
Games	**10**	**20**	**40**	**80**	**100**
Player 1 wins	4	15	21	39	47
Player 2 wins	6	5	19	41	53
Draws	0	0	0	0	0
Avg Time	11777.9	11010.55	12215.625	11999.8	17614.63

Gomoku MCTS vs MCTS - Comparison Across Iterations

Win Percentage View

Win Count View

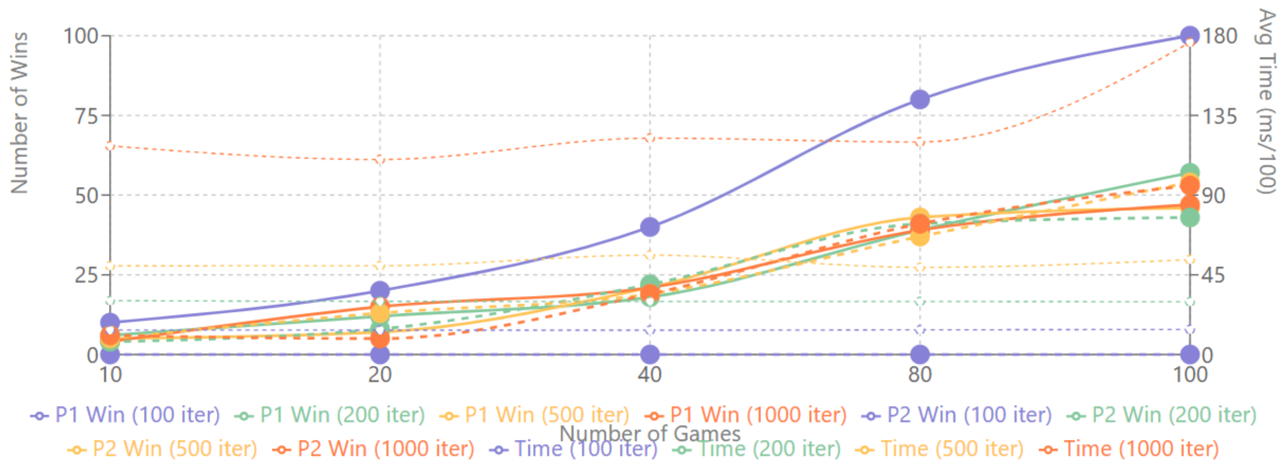


Note: Solid lines represent Player 1 wins, dashed lines represent Player 2 wins.
Time values are scaled down (divided by 100) to fit on the chart. See tooltip for actual values.

Gomoku MCTS vs MCTS - Comparison Across Iterations

Win Percentage View

Win Count View

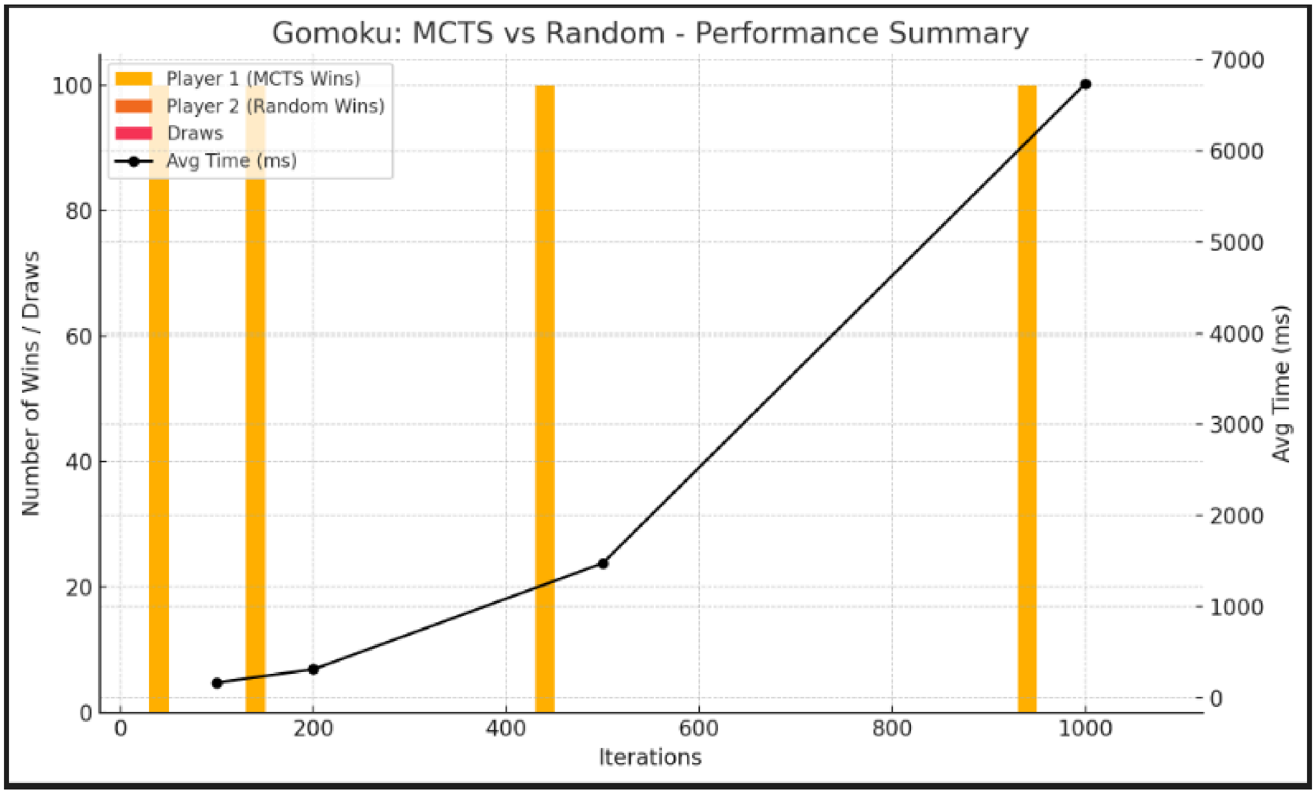


Note: Solid lines represent Player 1 wins, dashed lines represent Player 2 wins.
Time values are scaled down (divided by 100) to fit on the chart. See tooltip for actual values.

Results with number of iterations as variable – MCTS vs Random

The following table summarizes the outcomes when MCTS played against a random move player with increasing number of iterations (100, 200, 500, 1000):

Iterations	Games Played	MCTS Wins	Random Wins	Draws	Avg Times(ms)
100	100	100	0	0	167.81
200	100	100	0	0	312.86
500	100	100	0	0	1477.2
1000	100	100	0	0	6733.86

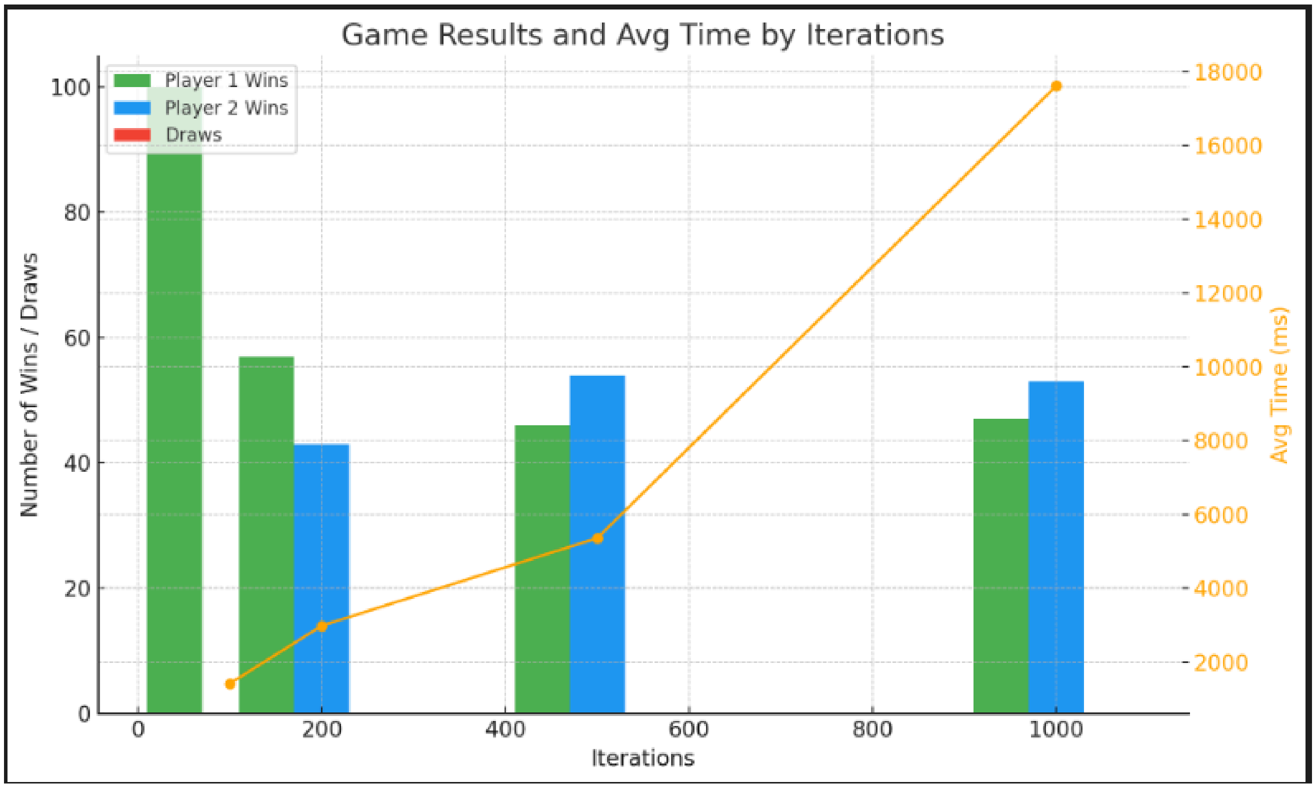


We observe that even at a relatively low iteration count (100), MCTS performs flawlessly against a random player. Increasing the number of iterations improves MCTS's confidence in its moves, but the win ratio remains constant (100%). However, this increase comes at the cost of significantly higher computational time. For instance, at 1000 iterations, the average runtime per game rose to approximately 6.7 seconds.

Results with MCTS vs MCTS

To further assess the algorithm, we examined how two MCTS agents performed against each other. Here, Player 1 always started first. This setup helped explore the impact of iterations on fairness and balance in the gameplay:

Iterations	Games Played	Player 1 Wins	Player 2 Wins	Draws	Avg Times(ms)
100	100	100	0	0	1412.25
200	100	57	43	0	2977.41
500	100	46	54	0	5354.37
1000	100	47	53	0	17614.63



At 100 iterations, Player 1 consistently won all games, suggesting a strong first-move advantage at lower iteration counts. However, as the number of iterations increased, the results began to balance out, with Player 2 winning almost an equal number of games at 500 and 1000 iterations. Despite this improvement in parity, draw occurrences remained at 0 across all configurations, possibly due to the deterministic nature of MCTS with identical settings.

Observations

- **Dominance vs Random**: MCTS convincingly outperforms random strategies even at low iteration counts.
- **Computation Cost**: Higher iteration values yield longer computation times. At 1000 iterations, average game times exceeded 17 seconds.
- **First Move Advantage**: When both players use MCTS, the player moving first has a significant advantage at lower iterations. This advantage decreases as iterations increase.
- **Draw Rate**: Surprisingly, no games ended in a draw, which could be attributed to Gomoku's design or to MCTS's deterministic decision-making.

Conclusion

- **Iteration Impact**: Increasing the number of iterations strengthens MCTS's decision-making but comes with a substantial increase in computation time. For Gomoku, while 100 iterations are enough to defeat a random player, at least 500–1000 iterations are required for fairer MCTS vs MCTS games.
- **First Move Bias**: The first move advantage is clearly present and diminishes only at higher iteration counts.

Citations:

http://vigir.missouri.edu/~gdesouza/Research/Conference_CDs/IEEE_SSCI_2016/pdf/SSCI16_paper_735.pdf

https://nestedsoftware.com/2019/08/07/tic-tac-toe-with-mcts-2h5k.152104.html
https://dl.acm.org/doi/10.5555/3466184.3466315