Program Structures and Algorithms

Spring 2024

Group Members: Siddharth Dumbre, Gaurav Gunjal
NUID: 002247119, 0022820367
GitHub Link: https://github.com/gunjalga/INFO6205_MCTS

Task: TicTacToe and DotsAndBoxes game implementation using MCTS

Abstract

This project explores the implementation of the Monte Carlo Tree Search (MCTS) algorithm in two classic 2-player games: Tic-Tac-Toe and Dots and Boxes. These games, known for their simplicity yet strategic depth, pose interesting challenges for strategic decision-making. In this study, we present an overview of both games and delve into the implementation details of MCTS to create game-playing agents capable of making strategic decisions.

Tic-Tac-Toe, a familiar game with a 3x3 grid, provides an ideal testing ground for exploring various decision-making techniques. Our implementation focuses on developing a player that employs MCTS to strategically explore the game tree, aiming to maximize its chances of winning.

Dots and Boxes played on a grid of dots, offer a more intricate strategic landscape. Here, our implementation harnesses the power of MCTS to navigate through possible moves, with the goal of constructing and capturing strategic boxes efficiently.

Monte Carlo Tree Search is a versatile algorithm that combines random simulation with tree-based exploration to make decisions in environments with large state spaces and complex decision trees. By leveraging MCTS, our implementations demonstrate adaptive and strategic gameplay, capable of providing engaging gaming experiences.

Through this project, we showcase the versatility of Monte Carlo Tree Search in developing agents for diverse gaming environments. Our implementations not only offer entertaining gameplay experiences but also serve as educational tools for understanding and exploring the potential of MCTS in game development and strategic decision-making.

TicTacToe Implementation

Tic-Tac-Toe is a classic 2-player game played on a 3x3 grid. The objective is to place three markers of the same kind (traditionally X or O) in a row, column, or diagonal.

Implementation Details:

Our implementation begins with the user providing the first move, which initializes the game board position. From this position, we create a state representation, encapsulating the current state of the game. This state is then used to create a node in the Monte Carlo Tree.

Monte Carlo Tree Search:

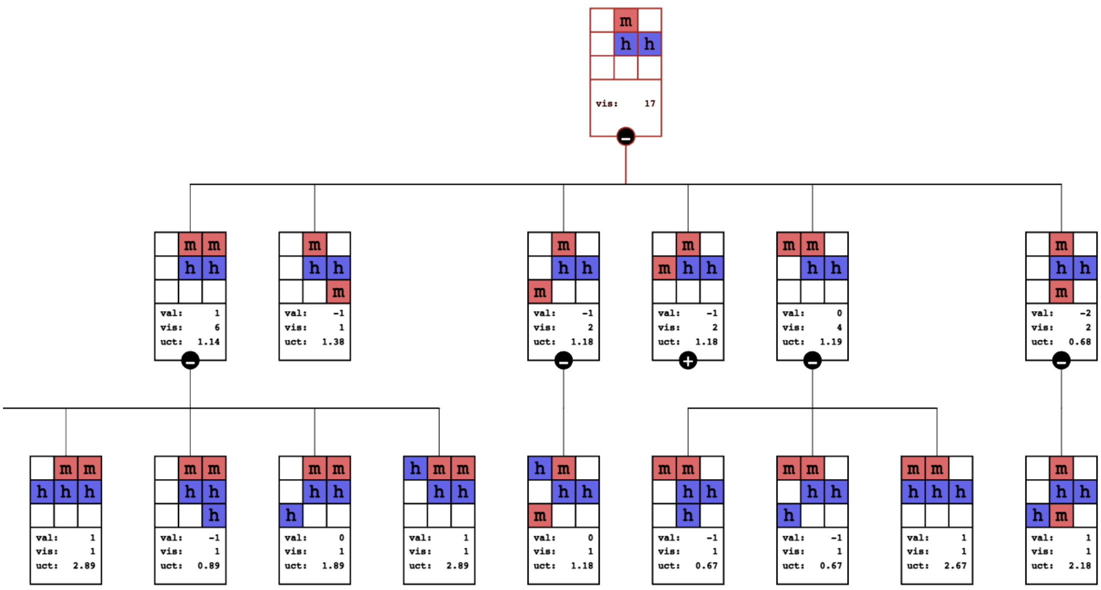
The MCTS algorithm is employed to make strategic decisions for the computer player. The process starts with expanding the current node by considering all possible single moves available to the computer. These moves are represented as child nodes of the current node.

Selection: The algorithm begins at the root node of a tree and proceeds to select subsequent child nodes. The selection is based on optimizing a specific criterion, typically the node with the highest win rate. The selection process uses the Upper Confidence Bound (UCB) formula, commonly referred to as the UCT (Upper Confidence bounds applied to Trees) in tree search contexts. This method balances exploration of less visited nodes and exploitation of nodes known to perform well. The selection phase continues until a leaf node, which has not yet been fully expanded, is reached.

Expansion: Once a leaf node is identified during the selection phase, the MCTS algorithm expands this node by adding all potential future states originating from it. This expansion adds these new nodes to the tree, each representing a possible future move or state in the game.

Simulation: After expansion, the algorithm performs a simulation starting from the new nodes. This simulation involves playing out the game from the node's state to a game conclusion using random moves, a process often referred to as a "playout". This phase is designed to provide a quick, randomized evaluation of the node’s potential without the need for deep computation.

Backpropagation: In the final phase, once the simulation reaches a conclusion (typically when the game ends), the results are used to update the tree. The algorithm backpropagates from the terminal state of the simulation up to the root node. During this process, it updates the statistics for each visited node, such as the win rate (if the simulation resulted in a win) and the visit count. This data helps in refining the selection strategy for future iterations of the algorithm.
MCTS iterates through these four phases repeatedly until a specified condition is met, such as a set number of iterations or a time limit, after which it selects the move leading to the node with the best performance metric as its decision.

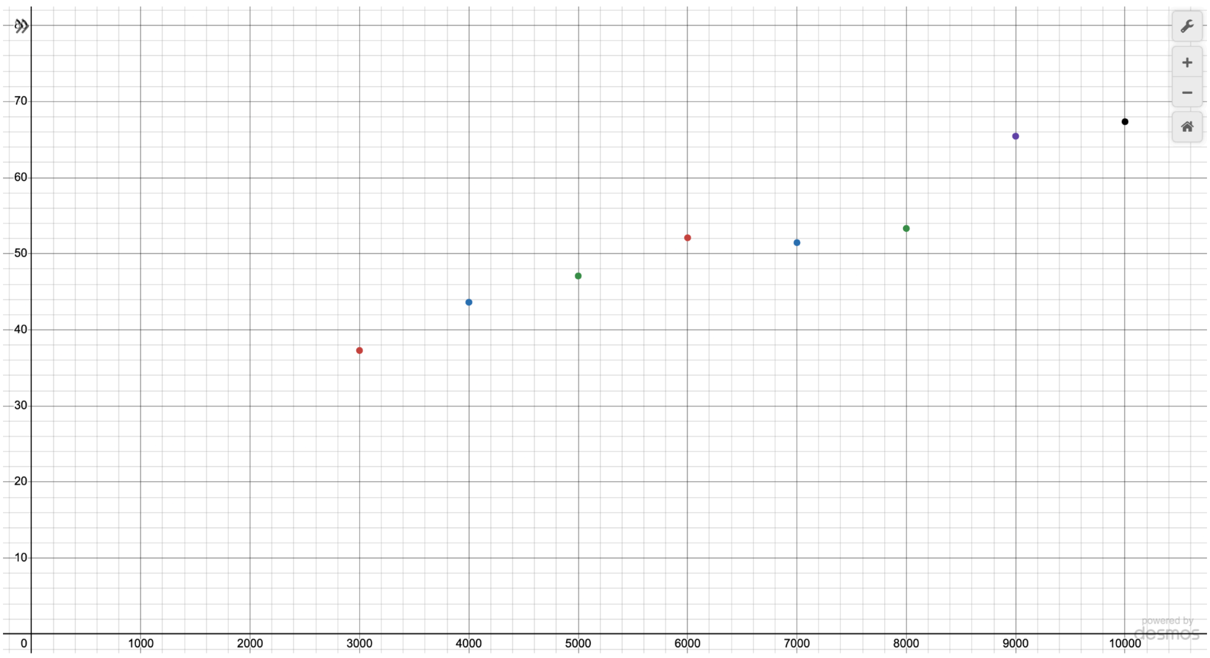


Tree formation based on the move made and all the available moves from that state

Impact of number of iterations

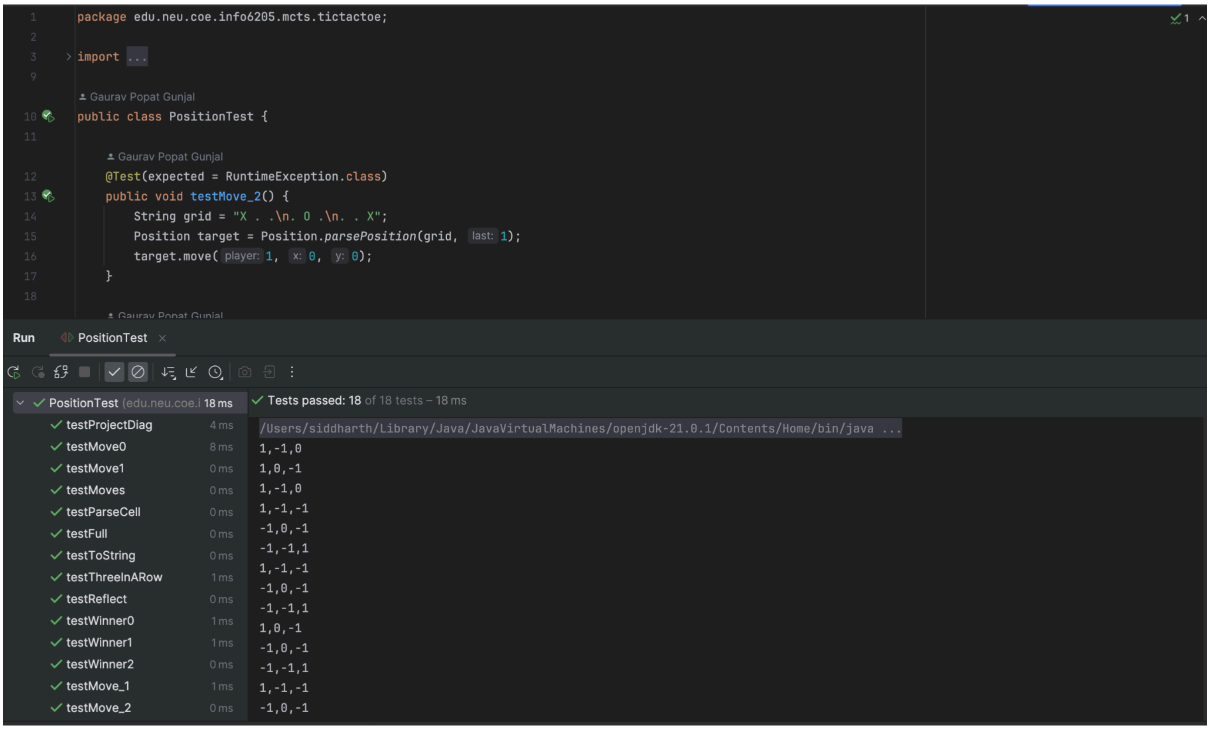
Increasing the number of iterations in the Monte Carlo Tree Search (MCTS) algorithm for our Tic-Tac-Toe game profoundly impacts its strategic decision-making process. With more iterations, the algorithm exhibits enhanced exploration capabilities, thoroughly analyzing a larger portion of the game tree. This expanded exploration enables the algorithm to make more informed decisions by considering a greater number of potential moves and their consequences. Additionally, as the number of iterations rises, the algorithm refines its move selection process, leading to more accurate assessments of move values and improved decision-making. Furthermore, the increased number of iterations allows the algorithm to accumulate more experience through simulated gameplay, facilitating enhanced learning and adaptation over time. As a result, increasing the number of iterations in the MCTS algorithm leads to improved performance, stronger gameplay, and convergence towards optimal decision-making strategies. However, it's important to note that this enhanced performance comes at the cost of increased computational time, as reflected in the timing comparisons provided in the accompanying table.

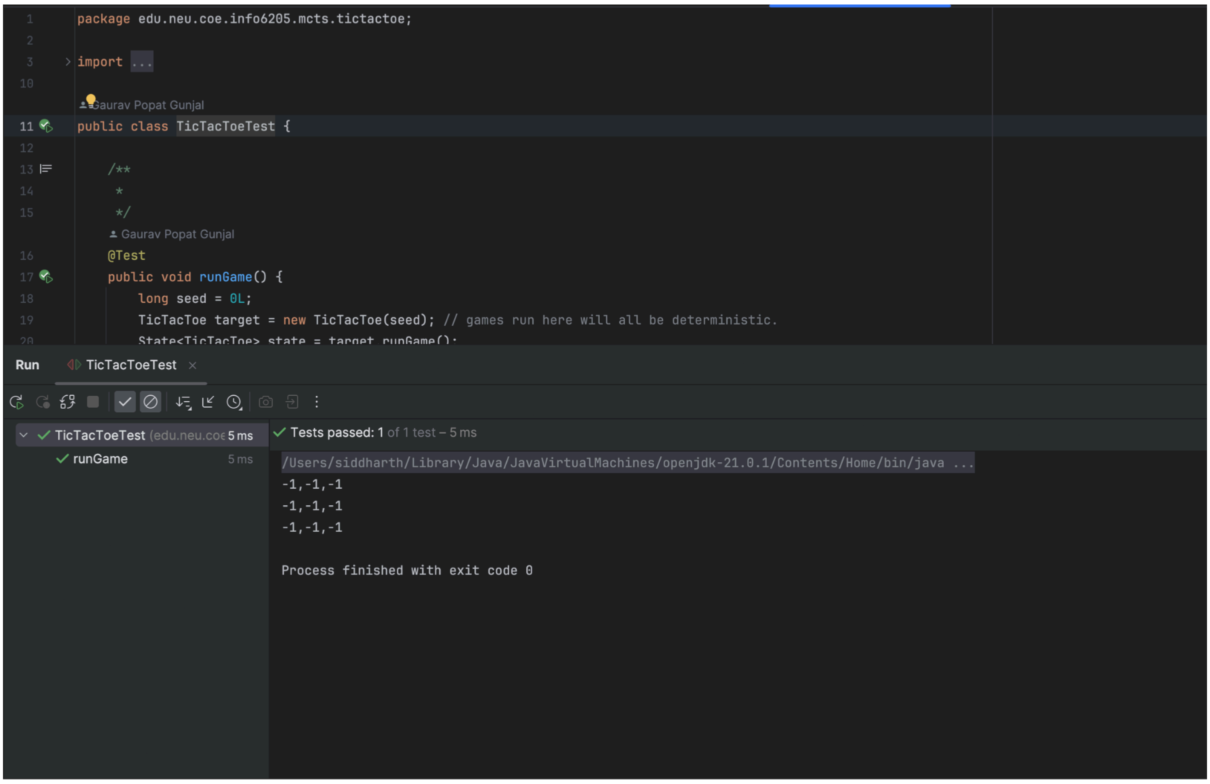
Number of iterations	Time taken
3000	37.29
4000	43.625
5000	47.08
6000	52.10275
7000	51.463
8000	53.33
9000	65.45279
10000	67.36

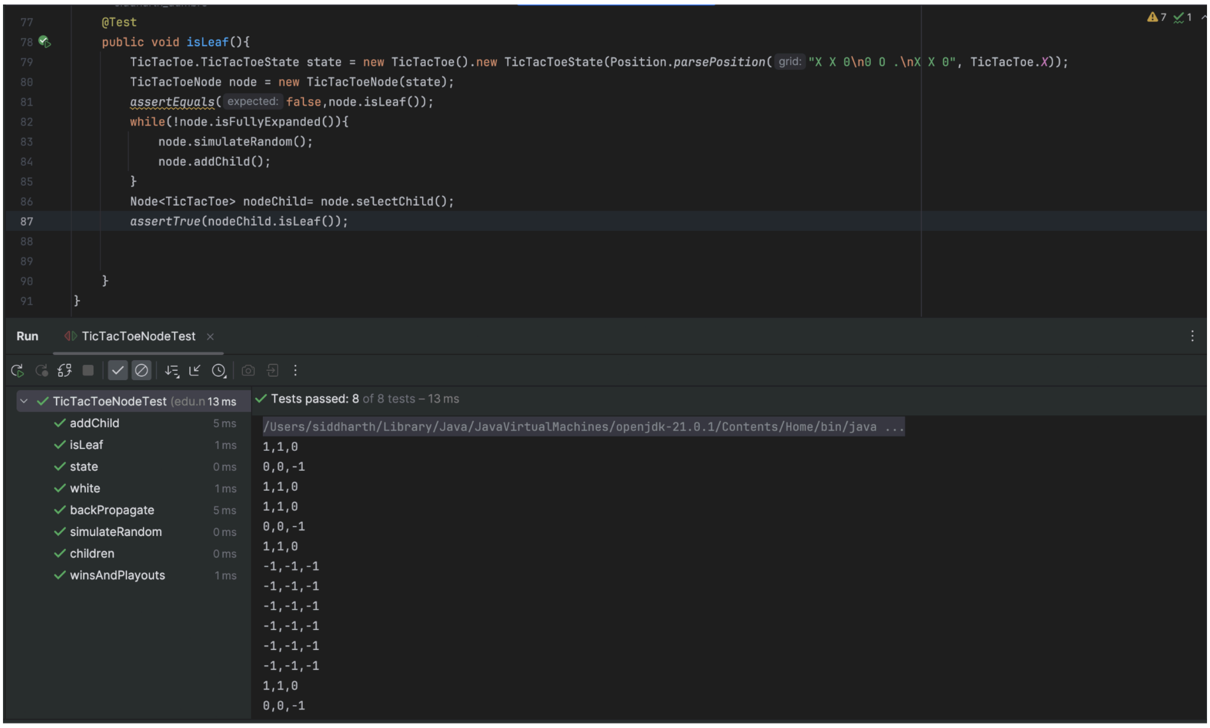


Time taken for a move(expansion+simulation+backpropagation) vs Number of Iterations

Test Cases


package edu.neu.coe.info6205.mcts.tictactoe;
import ...
Gaurav Popat Gunjal
public class PositionTest {
Gaurav Popat Gunjal
@Test(expected = RuntimeException.class)
public void testMove_2() {
String grid = "X . .\n. 0 .\n. . X";
Position target = Position.parsePosition(grid, last: 1);
target.move(player: 1, x: 0, y: 0);
}
Run
PositionTest
PositionTest (edu.neu.coe.i 18 ms
testProjectDiag 4 ms
testMove0 8 ms
testMove1 0 ms
testMoves 0 ms
testParseCell 0 ms
testFull 0 ms
testToString 0 ms
testThreeInARow 1 ms
testReflect 0 ms
testWinner0 1 ms
testWinner1 1 ms
testWinner2 0 ms
testMove_1 1 ms
testMove_2 0 ms
Tests passed: 18 of 18 tests – 18 ms
/Users/siddharth/Library/Java/JavaVirtualMachines/openjdk-21.0.1/Contents/Home/bin/java ...
1,-1,0
1,0,-1
1,-1,0
1,-1,-1
-1,0,-1
-1,-1,1
1,-1,-1
-1,0,-1
-1,-1,1
1,0,-1
-1,0,-1
-1,-1,1
1,-1,-1
-1,0,-1


package edu.neu.coe.info6205.mcts.tictactoe;
import ...
Gaurav Popat Gunjal
public class TicTacToeTest {
/**
*
*/
Gaurav Popat Gunjal
@Test
public void runGame() {
long seed = 0L;
TicTacToe target = new TicTacToe(seed); // games run here will all be deterministic.
Run
TicTacToeTest
TicTacToeTest (edu.neu.coe 5 ms
runGame 5 ms
Tests passed: 1 of 1 test – 5 ms
/Users/siddharth/Library/Java/JavaVirtualMachines/openjdk-21.0.1/Contents/Home/bin/java ...
-1,-1,-1
-1,-1,-1
-1,-1,-1
Process finished with exit code 0



```java
    @Test
    public void isLeaf(){
        TicTacToe.TicTacToeState state = new TicTacToe().new TicTacToeState(Position.parsePosition( grid: "X X 0\n0 0 .\nX X 0", TicTacToe.X));
        TicTacToeNode node = new TicTacToeNode(state);
        assertEquals( expected: false,node.isLeaf());
        while(!node.isFullyExpanded()){
            node.simulateRandom();
            node.addChild();
        }
        Node<TicTacToe> nodeChild= node.selectChild();
        assertTrue(nodeChild.isLeaf());


    }
}
```
Run
TicTacToeNodeTest
TicTacToeNodeTest (edu.n 13 ms
addChild 5 ms
isLeaf 1 ms
state 0 ms
white 1 ms
backPropagate 5 ms
simulateRandom 0 ms
children 0 ms
winsAndPlayouts 1 ms
Tests passed: 8 of 8 tests – 13 ms
/Users/siddharth/Library/Java/JavaVirtualMachines/openjdk-21.0.1/Contents/Home/bin/java ...
1,1,0
0,0,-1
1,1,0
1,1,0
0,0,-1
1,1,0
-1,-1,-1
-1,-1,-1
-1,-1,-1
-1,-1,-1
-1,-1,-1
-1,-1,-1
1,1,0
0,0,-1

Dots and Boxes Implementation

Data structure: We are using an n*n grid of Box objects, each of these box objects will have 4 booleans left, right, top, and bottom to represent the edges of a box, and x, and y variables to help us determine the position of the box on the grid.

This n*n grid is stored in a BoxPosition object which contains methods for operations on the grid like making a move, getting all the possible moves from a position, checking if the grid is full, getting a winner, etc.

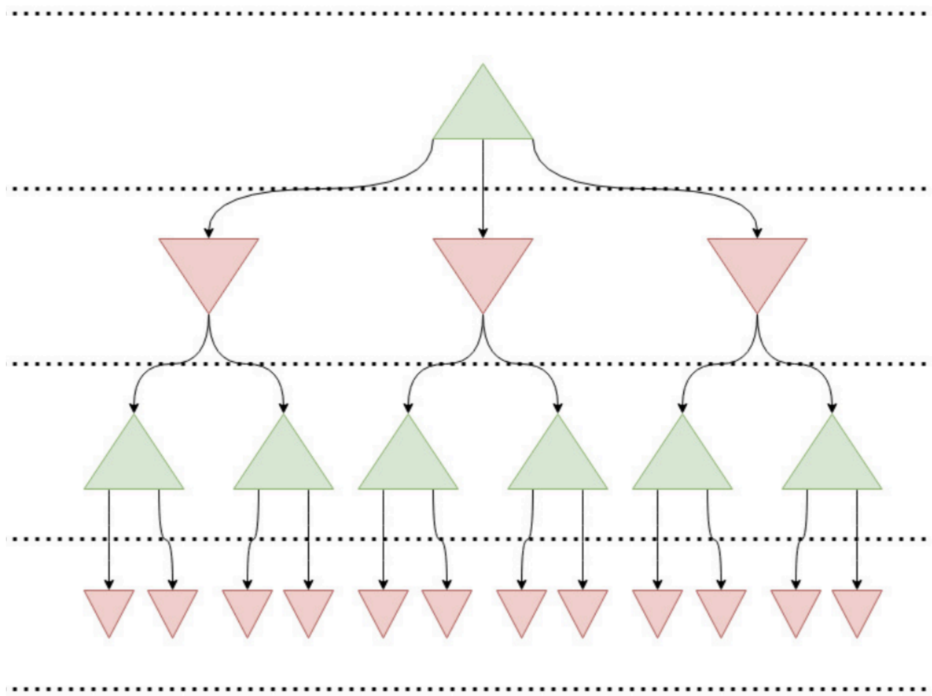
This data structure might be very simple to understand but it has its drawbacks

Keeping in mind that we cannot use the same references of box objects in different nodes as it would cause a clash of operations on the same references leading to the failure of the MCTS algorithm, we created a copy grid method which gives a new instance of every single box object in the position.

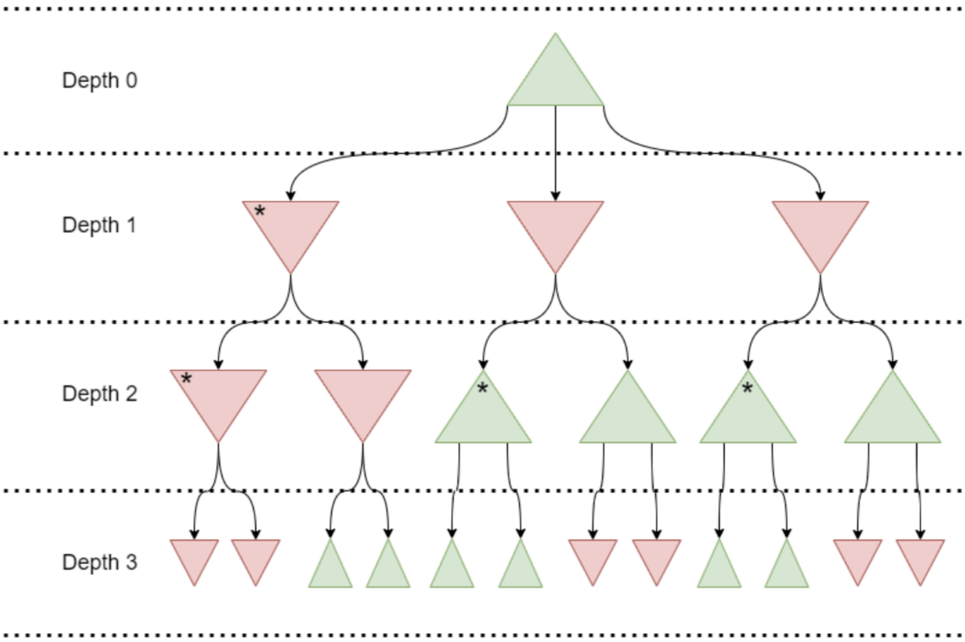
While making a move we need to handle the booleans of two neighboring box objects to make sure everything on the grid is consistent.

Moves: Unlike the Tic-tac-toe move, here we need to take three parameters from the user that is the x,y coordinate of the box and the direction(which will help us determine which edge in x,y box)

MCTS for Dots and Boxes: In the MCTS implementation for dots and boxes the selection, expansion, and simulation phase are quite similar to the tic-tac-toe game but the backpropagation phase is different which makes the algorithm quite complex. As we know in a tic-tac-toe game the players take turns in one after the other fashion but in the dots and boxes, a player can get a chain of turns if he continues to capture a box. So we have to keep track of the player to maximize at each node and the nodes below that node. In games like tic-tac-toe, we move in min-max-min-max fashion but here we might have to change to order to let's say min-max-max-max-min if a player captures 3 boxes in 3 consecutive moves.



Min-max for tic-tac-toe



Min-max for dots and boxes. * represents a state where box is captured

Finding balance between exploration and exploitation terms:

This is the formula to calculate UCT score for a particular node. There are two terms in this formula. The left hand side term is the exploitation term and the right hand side term is the exploration term. And c is UCB constant which will determine how much the algorithm will explore from all available nodes at a level in the tree.

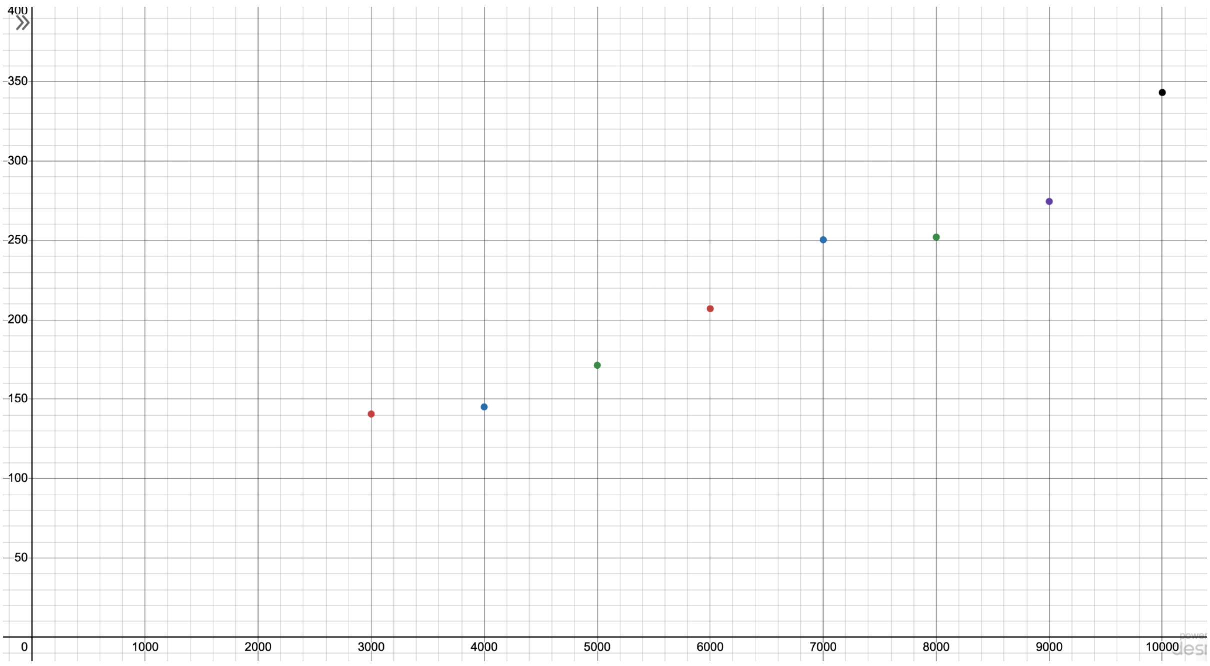
$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}}$$

- w_i stands for the number of wins for the node considered after the i-th move
- n_i stands for the number of simulations for the node considered after the i-th move
- N_i stands for the total number of simulations after the i-th move run by the parent node of the one considered
- c is the exploration parameter—theoretically equal to $\sqrt{2}$; in practice usually chosen empirically

In a 3*3 grid for dots and boxes we have 9 total boxes and 24 different edges, the total nodes after a single move are around 4048. To ensure that the algorithm doesn't converge on one single node as the tree below that node reaches its leaf levels, we need to increase the UCB constant so that our algorithm goes down each path and tries to find an optimal path. Also we need to keep the number of iterations on the higher side for a game like dots and boxes as the number of possible nodes from one node are quite large.

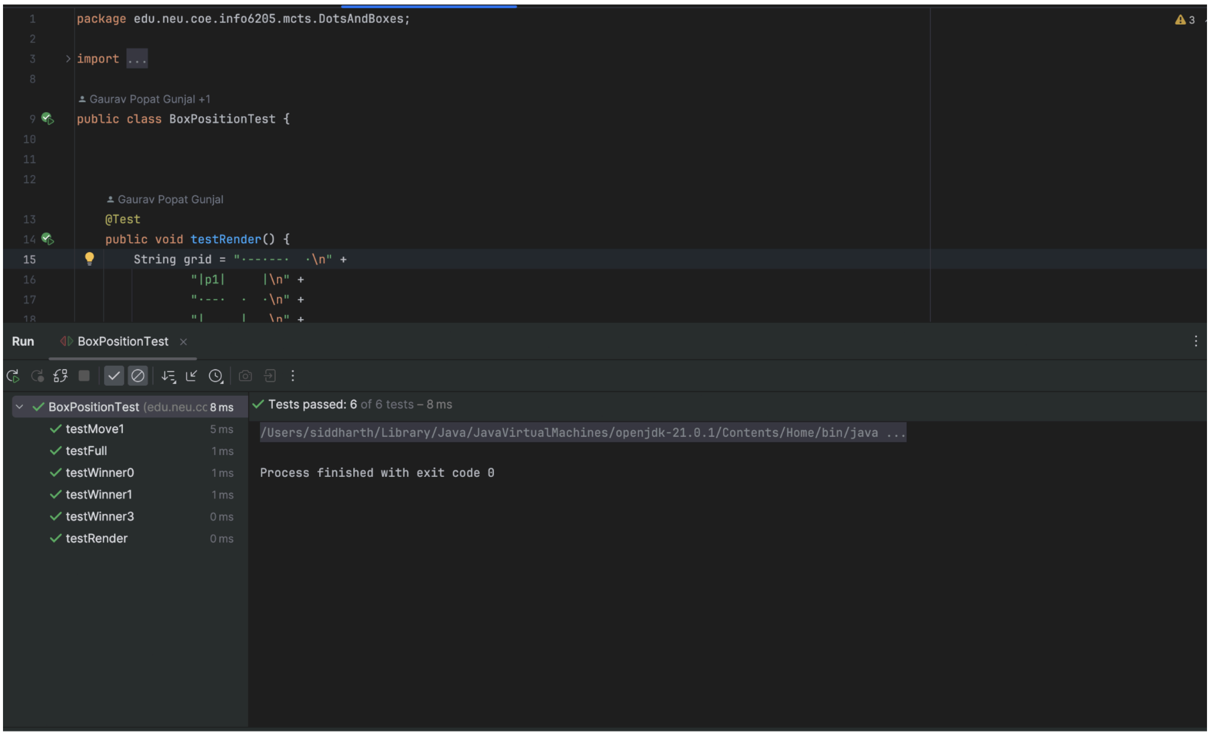
What else could we do to optimise the algorithm? Instead of assigning the nodes with +1 or -1 for winning and losing simulation, could we just use the number of boxes captured as the value for wins? Well if you think about it carefully it doesn't really have any effect on the algorithm at all because it will do the same for the levels of the opposite player as well. This makes it almost the same as adding just +1 and -1 to the nodes.

No of Iterations	Time taken(ms)
3000	140.7242
4000	145.19
5000	171.422
6000	207.0541
7000	250.48
8000	252.18
9000	274.61
10000	343.27

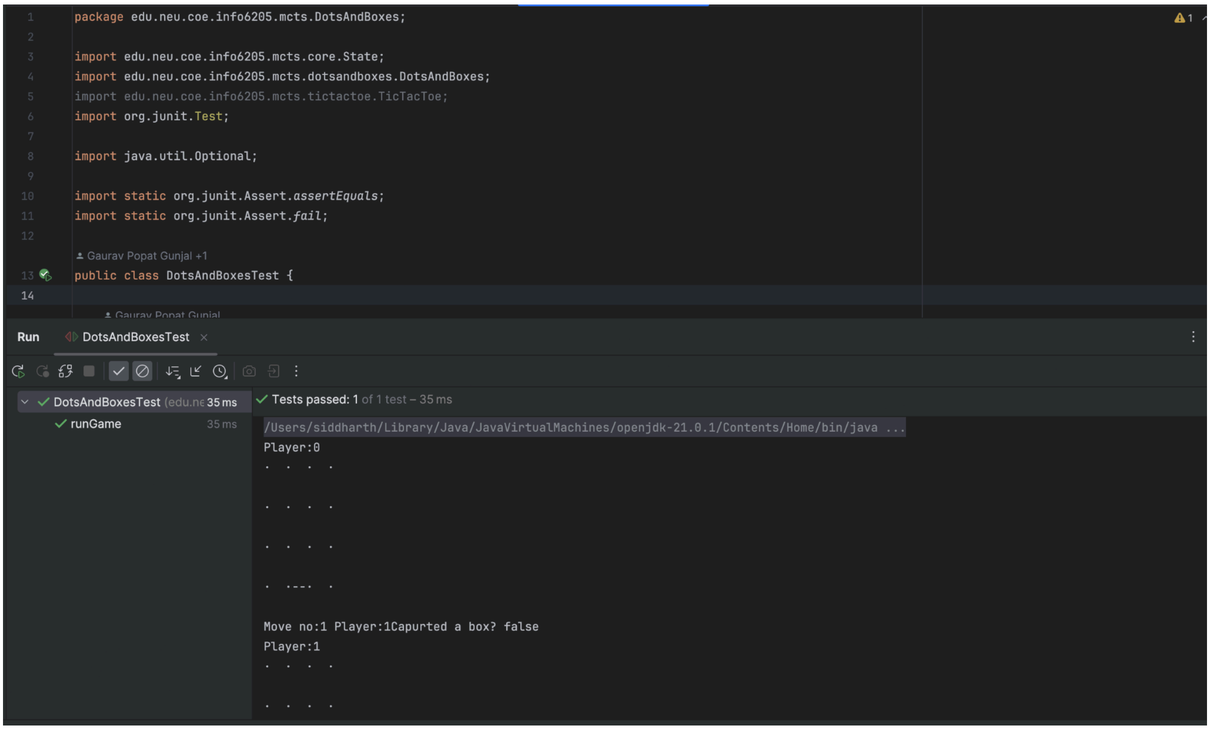


Time taken for a move(expansion+simulation+backpropagation+selection) vs Number of iterations

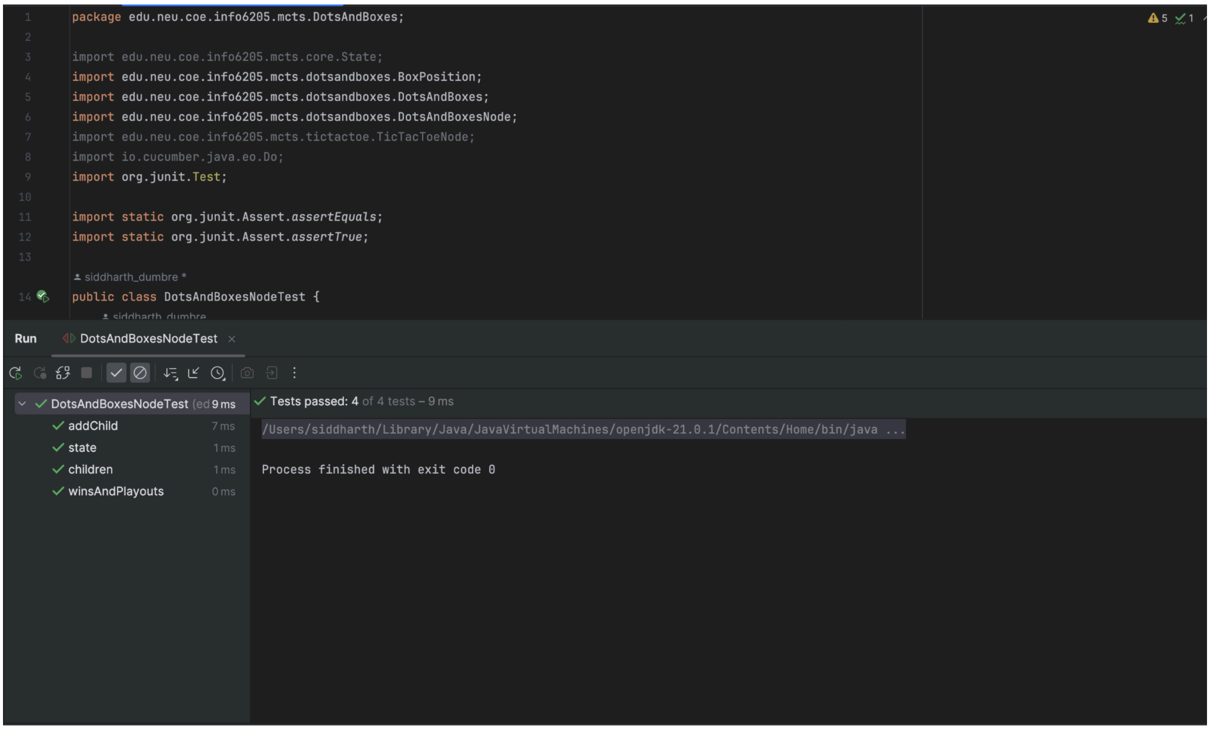
Test Cases



package edu.neu.coe.info6205.mcts.DotsAndBoxes;
import ...
Gaurav Popat Gunjal +1
public class BoxPositionTest {
Gaurav Popat Gunjal
@Test
public void testRender() {
String grid = "·------- ·\n" +
"|p1| |\n" +
"·---- · ·\n" +
Run
BoxPositionTest
BoxPositionTest (edu.neu.cc 8 ms
testMove1 5 ms
testFull 1 ms
testWinner0 1 ms
testWinner1 1 ms
testWinner3 0 ms
testRender 0 ms
Tests passed: 6 of 6 tests – 8 ms
/Users/siddharth/Library/Java/JavaVirtualMachines/openjdk-21.0.1/Contents/Home/bin/java ...
Process finished with exit code 0



package edu.neu.coe.info6205.mcts.DotsAndBoxes;
import edu.neu.coe.info6205.mcts.core.State;
import edu.neu.coe.info6205.mcts.dotsandboxes.DotsAndBoxes;
import edu.neu.coe.info6205.mcts.tictactoe.TicTacToe;
import org.junit.Test;
import java.util.Optional;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.fail;
Gaurav Popat Gunjal +1
public class DotsAndBoxesTest {
Run
DotsAndBoxesTest
DotsAndBoxesTest (edu.ne 35 ms
runGame 35 ms
Tests passed: 1 of 1 test – 35 ms
/Users/siddharth/Library/Java/JavaVirtualMachines/openjdk-21.0.1/Contents/Home/bin/java ...
Player:0
Move no:1 Player:1Capurted a box? false
Player:1



Conclusion

After implementation of the MCTS algorithm for the TicTacToe and DotsAndBoxes games, it was observed that the time taken for the computer to make a move in the DotsAndBoxes game was much higher compared to TicTacToe. The reason for this is that the number of possible moves in the DotsAndBoxes game is much higher than tictactoe because it comprises all the edges and if a box is captured then the same player gets a new chance. For this reason, the computer evaluates the most optimal move based on the consecutive and non-consecutive moves depending on whether the box is captured by that move or not. Also, it is observed that as the number of iterations increases by a step of 1000 then the timing for the computer to make a move increases. Increasing the number of iterations in the Monte Carlo Tree Search algorithm increases computational time because each iteration involves additional simulations and evaluations of potential moves, leading to a proportional increase in the overall computational workload.

References

https://vgarciasc.github.io/mcts-viz/
https://www.youtube.com/watch?v=UXW2yZndl7U&t=412s
https://pats.cs.cf.ac.uk/@archive_file?p=1686&n=final&f=1-report.pdf&SIG=4eec909fce4fb2da94c4b05d1906f909face6aa1dc12dd5def36b33ff165eace