Program Structures and Algorithms
Spring 2025

NAME: Mingrui Yuan, Yuanchang Lee, Ronghao Yuan
NUID: 002089150, 002310611, 002086973
GITHUB LINK: https://github.com/info6206final/finalProject6205

Executive Summary

For this project, our team implemented a basic Monte Carlo Tree Search (MCTS) agent to play the game of Go. The idea was to create an AI that can play against a human player using the MCTS algorithm and integrate it into a graphical Go game. We used Java and added our code into the existing project structure, combining game logic with a user interface built using Swing. The final result is a playable Go game where a human can compete against an AI opponent. The AI calculates moves using MCTS and responds right after the player's move.

Introduction to The Game of Go

Go is a two-player board game that dates back over 2,500 years. It's usually played on a 19×19 grid—smaller 9×9 or 13×13 boards are common when learning the game.

Players take turns placing black or white stones on empty intersections. A stone or a connected group of stones is captured and removed if all its adjacent empty points (liberties) are occupied by the opponent.

The goal is to control more territory than your opponent. Territory consists of empty intersections you've surrounded with your stones. When both players pass in succession, play ends. Each scores by counting controlled territory plus captured stones; the "Ko" rule prevents infinite repetition by forbidding a move that would recreate the previous board position.

Project Structure

We built our project under this package:
com.phasmidsoftware.dsaipg.projects.mcts.gogame.src

Main files we worked on:

1. **Chess_B.java**: A class for storing individual piece data.
2. **ChessBoardPanel_B.java**: Handles game logic, the board state, and all the rules.
3. **GoGameGUI_B.java**: A graphical interface for interacting with the game.
4. **MCSTAgent.java:** Our main Monte Carlo Tree Search implementation.
5. **Memeryunit_B.java**: A helper class for storing board history (used for undo moves).

Go Game Implementation

We built a playable Go game using Java Swing. The board is 19x19, and users can place stones by clicking on the board. The game handles basic rules like:

- Valid move detection
- Capturing stones based on liberties
- Group formation and life-death detection

Key Implementation Details

1.State
Encapsulates the board (19×19 array) and current player.

- getLegalMoves() returns all empty positions.
- applyMove() clones the board, plays a stone, and switches turn.
- evaluate() compares black vs. white stone counts.

2.TreeNode
Holds a State, pointers to parent/children, counts (visits, wins), and untried moves.

- getUCTValue() computes the UCT score.
- selectChild() picks the child with maximum UCT.

3.Simulation Depth & Iterations

- **ITERATION_LIMIT** = 4000 playouts
- **MAX_SIMULATION_DEPTH** = 100 moves

Go's true win/loss determination and endgame require intricate territory scoring, capture tracking, and proper handling of ko fights. Implementing a full, rules-compliant scoring engine is a major undertaking. This project uses only a stone-count heuristic (black vs. white stones) and does *not* address ko rule enforcement or advanced endgame evaluation. As a result, the AI's playing strength remains very limited.

Algorithm Design and Implementation

In this project, we used the Monte Carlo Tree Search (MCTS) algorithm to build an AI that can play Go against a human. Since Go is a complex game, especially on a 19x19 board, we decided to start with a simplified version of the rules — we focus only on placing stones, capturing, and counting pieces at the end. We didn't include things like Ko or full territory scoring yet.

How the MCTS Agent Works

The idea behind MCTS is to simulate lots of random games and see which moves lead to better outcomes. For each AI move, the program goes through four steps:

1. **Selection**:Starting from the current board state, the algorithm selects child nodes using the UCT (Upper Confidence Bound for Trees) formula to balance exploration and exploitation. It continues down the tree until it reaches a node that hasn't been fully explored.

$$\mathrm{UCT} = \frac{\mathrm{wins}}{\mathrm{visits}} + C\sqrt{\frac{\ln(\mathrm{parent.visits})}{\mathrm{visits}}}$$

2. **Expansion**: When it finds a move it hasn't tried yet, it adds it to the tree.

3. **Simulation**: From that new move, it plays out a random game until either the board is full or it reaches a maximum depth (we set it to 100 moves). It counts how many black and white pieces are on the board to see who would be ahead.

4. **Backpropagation**:The result of that simulation (black win, white win, or draw) is sent back up the tree, and all the nodes that led to that move get updated.

After running this process 4000 times for each AI move, the AI picks the move that was the most successful during the simulations.

Game State and Move Logic

The board is represented as a 19x19 integer matrix. We use:

- 0 for empty spaces,

- 1 for black stones,
- -1 for white stones.

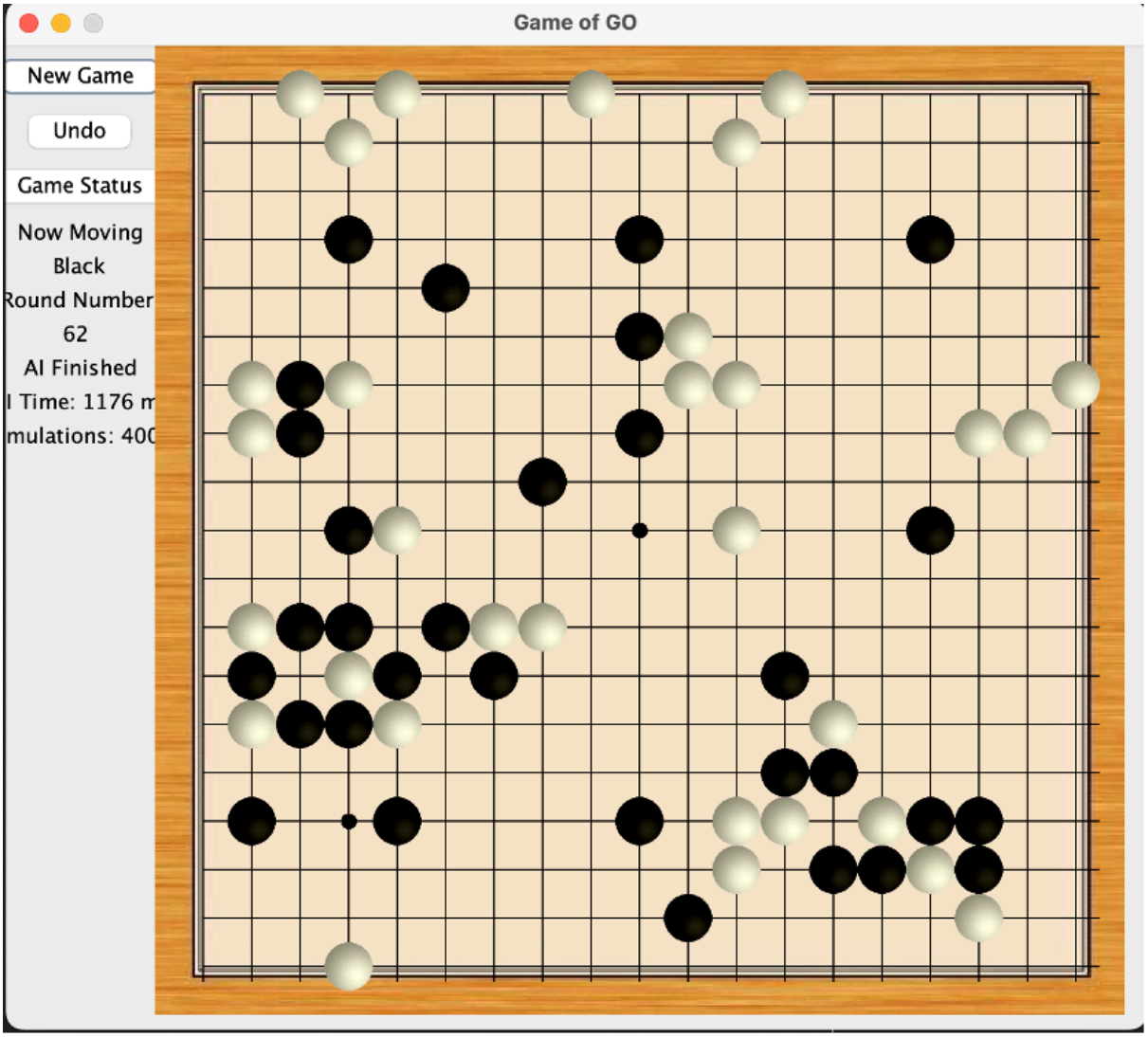
Each move creates a new board state and switches turns. For performance reasons, we simplified the simulation phase by skipping full liberty/capture checking, and instead just placing stones randomly. This makes it possible to run many simulations quickly while still getting reasonable results.

Parameters We Used

- UCT exploration constant: **1.41**
- Simulations per move: **4000**
- Max moves per simulation: **100**

How AI Plays the Game

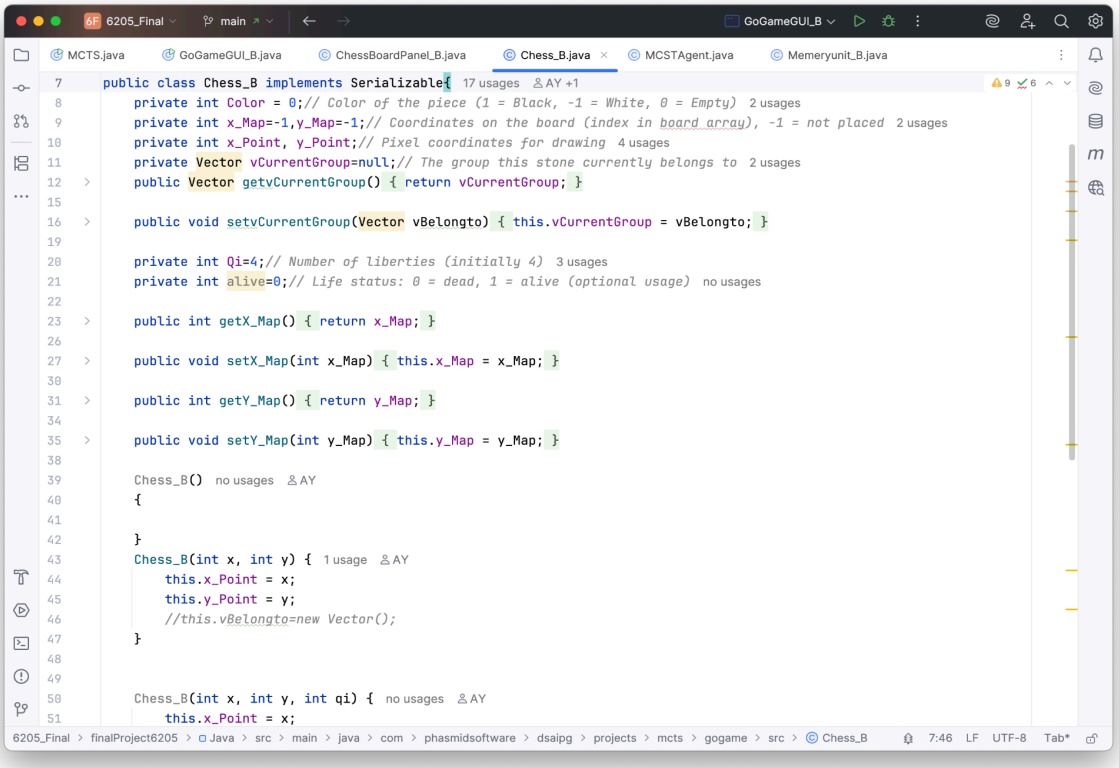
We modified the GUI code so that when the player makes a move, the AI will immediately take its turn using the MCTS algorithm. The steps are: Get the current board state. Run MCSTAgent.nextMove() to find the best move. Use playChessAI() to make the move on the board. Update the interface with the new move.



Test Cases

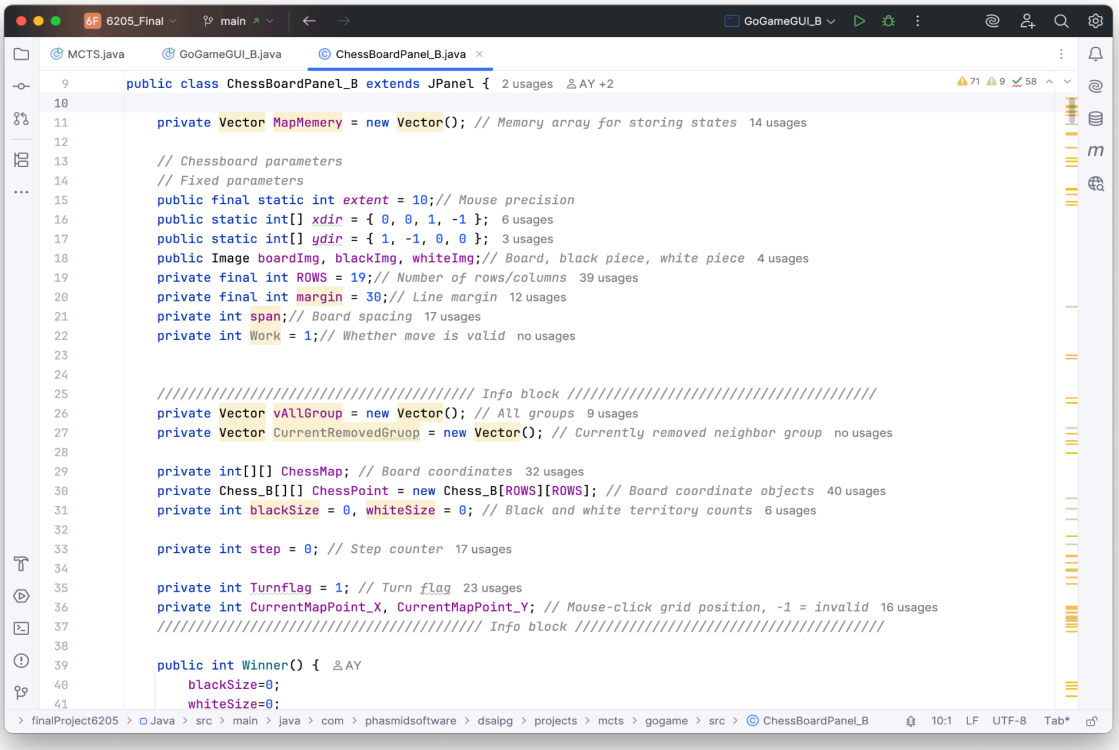
In the “GoGameTestCases.java” class, we have several test cases in order to test the functionality of our game. To be more specific, we used setUp() to initialize a new game, resetting the board and step count. TestNewGameState1() calls newGame(), and then verifies whether the initial state of the game is correct. The testInvalidMove() class tries to access an invalid location of the board that should be returning a 0. And the testMoveSurround() class tests an intentionally suicidal move where it would be surrounded by the opponent. The expected output of testMoveSurround() is a -1. Last but not least, testCapture() is trying to surround a piece to capture it. Then we verify if the piece at position [1][1] becomes 0.

Code Screenshot



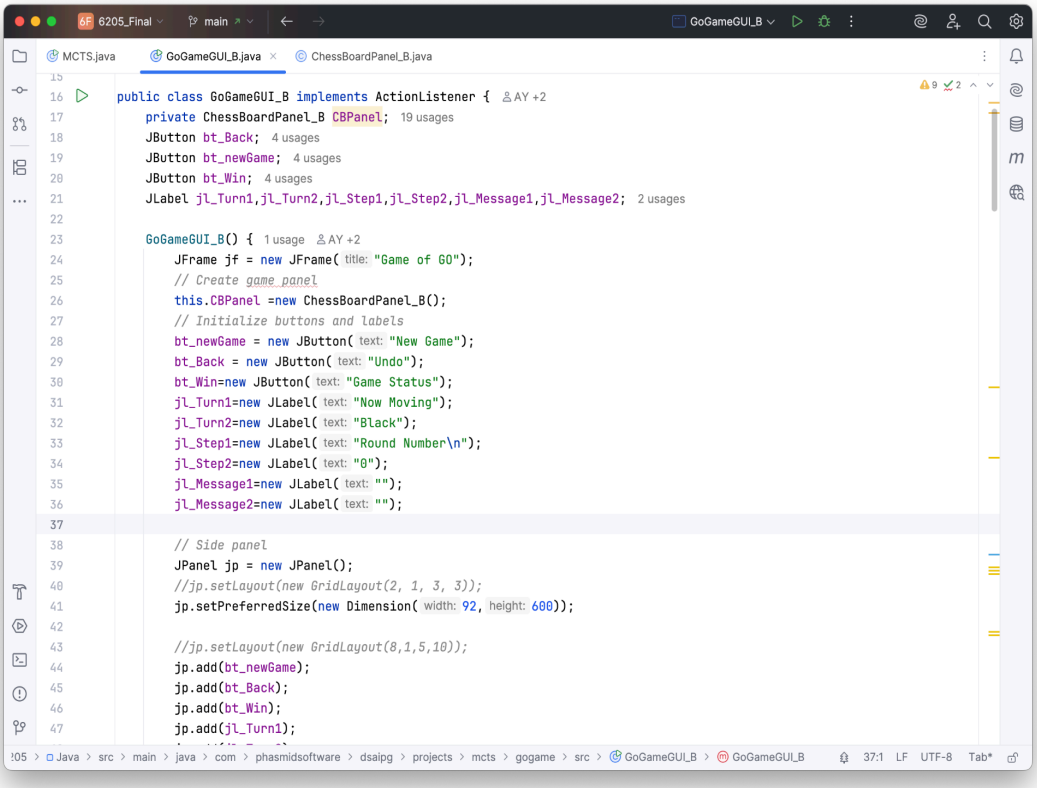
```java
public class Chess_B implements Serializable{ 17 usages  AY +1
    private int Color = 0;// Color of the piece (1 = Black, -1 = White, 0 = Empty)  2 usages
    private int x_Map=-1,y_Map=-1;// Coordinates on the board (index in board array), -1 = not placed  2 usages
    private int x_Point, y_Point;// Pixel coordinates for drawing  4 usages
    private Vector vCurrentGroup=null;// The group this stone currently belongs to  2 usages
    public Vector getvCurrentGroup() { return vCurrentGroup; }

    public void setvCurrentGroup(Vector vBelongto) { this.vCurrentGroup = vBelongto; }

    private int Qi=4;// Number of liberties (initially 4)  3 usages
    private int alive=0;// Life status: 0 = dead, 1 = alive (optional usage)  no usages

    public int getX_Map() { return x_Map; }

    public void setX_Map(int x_Map) { this.x_Map = x_Map; }

    public int getY_Map() { return y_Map; }

    public void setY_Map(int y_Map) { this.y_Map = y_Map; }

    Chess_B()  no usages  AY
    {

    }
    Chess_B(int x, int y) {  1 usage  AY
        this.x_Point = x;
        this.y_Point = y;
        //this.vBelongto=new Vector();
    }


    Chess_B(int x, int y, int qi) {  no usages  AY
        this.x_Point = x;
```

(Chess_B.java)



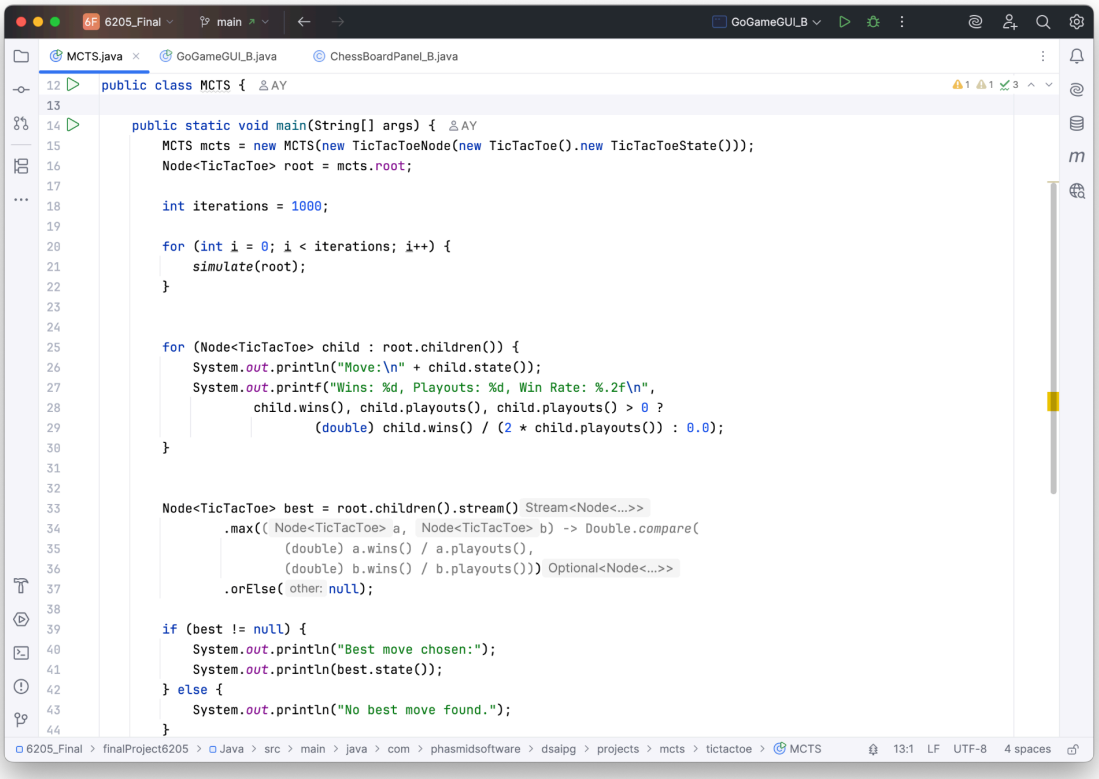
```java
public class ChessBoardPanel_B extends JPanel {  2 usages  AY +2

    private Vector MapMemery = new Vector(); // Memory array for storing states  14 usages

    // Chessboard parameters
    // Fixed parameters
    public final static int extent = 10;// Mouse precision
    public static int[] xdir = { 0, 0, 1, -1 };  6 usages
    public static int[] ydir = { 1, -1, 0, 0 };  3 usages
    public Image boardImg, blackImg, whiteImg;// Board, black piece, white piece  4 usages
    private final int ROWS = 19;// Number of rows/columns  39 usages
    private final int margin = 30;// Line margin  12 usages
    private int span;// Board spacing  17 usages
    private int Work = 1;// Whether move is valid  no usages


    ////////////////////////////////////// Info block //////////////////////////////////////
    private Vector vAllGroup = new Vector(); // All groups  9 usages
    private Vector CurrentRemovedGroup = new Vector(); // Currently removed neighbor group  no usages

    private int[][] ChessMap; // Board coordinates  32 usages
    private Chess_B[][] ChessPoint = new Chess_B[ROWS][ROWS]; // Board coordinate objects  40 usages
    private int blackSize = 0, whiteSize = 0; // Black and white territory counts  6 usages

    private int step = 0; // Step counter  17 usages

    private int Turnflag = 1; // Turn flag  23 usages
    private int CurrentMapPoint_X, CurrentMapPoint_Y; // Mouse-click grid position, -1 = invalid  16 usages
    ////////////////////////////////////// Info block //////////////////////////////////////

    public int Winner() {  AY
        blackSize=0;
        whiteSize=0;
```

(ChessBoardPanel_B.java)



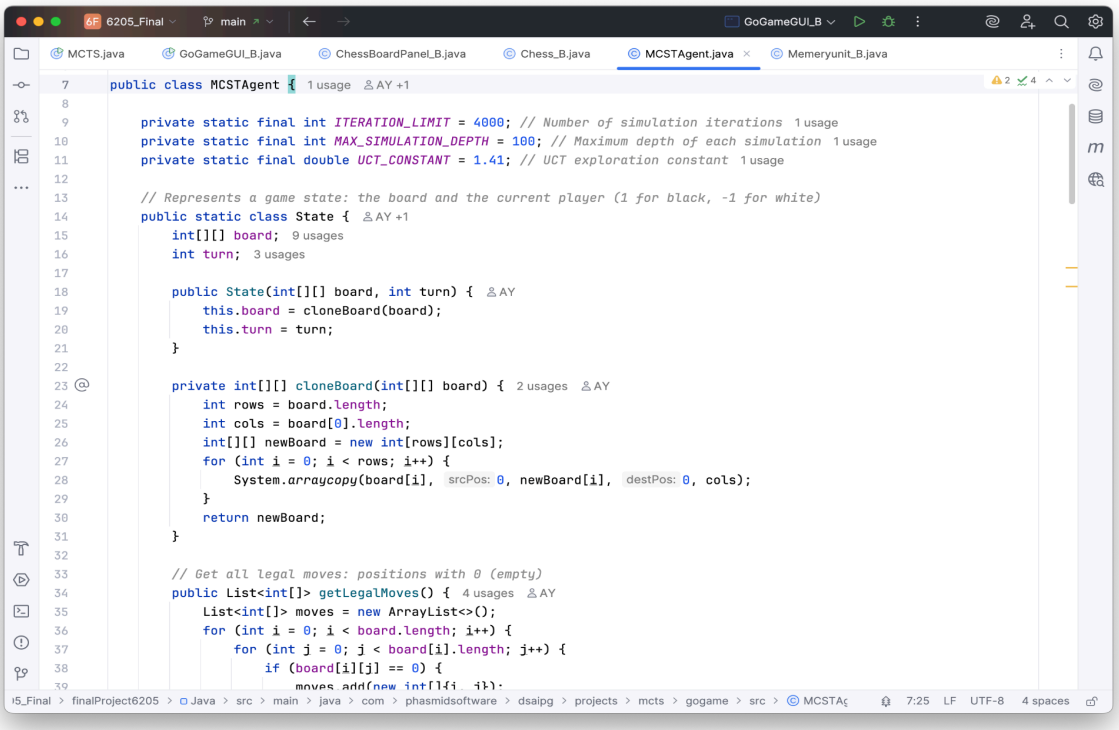
```java
public class GoGameGUI_B implements ActionListener {  AY +2
    private ChessBoardPanel_B CBPanel;  19 usages
    JButton bt_Back;  4 usages
    JButton bt_newGame;  4 usages
    JButton bt_Win;  4 usages
    JLabel jl_Turn1,jl_Turn2,jl_Step1,jl_Step2,jl_Message1,jl_Message2;  2 usages

    GoGameGUI_B() {  1 usage  AY +2
        JFrame jf = new JFrame( title: "Game of GO");
        // Create game panel
        this.CBPanel =new ChessBoardPanel_B();
        // Initialize buttons and labels
        bt_newGame = new JButton( text: "New Game");
        bt_Back = new JButton( text: "Undo");
        bt_Win=new JButton( text: "Game Status");
        jl_Turn1=new JLabel( text: "Now Moving");
        jl_Turn2=new JLabel( text: "Black");
        jl_Step1=new JLabel( text: "Round Number\n");
        jl_Step2=new JLabel( text: "0");
        jl_Message1=new JLabel( text: "");
        jl_Message2=new JLabel( text: "");

        // Side panel
        JPanel jp = new JPanel();
        //jp.setLayout(new GridLayout(2, 1, 3, 3));
        jp.setPreferredSize(new Dimension( width: 92, height: 600));

        //jp.setLayout(new GridLayout(8,1,5,10));
        jp.add(bt_newGame);
        jp.add(bt_Back);
        jp.add(bt_Win);
        jp.add(jl_Turn1);
```

(GoGameGUI_B.java)

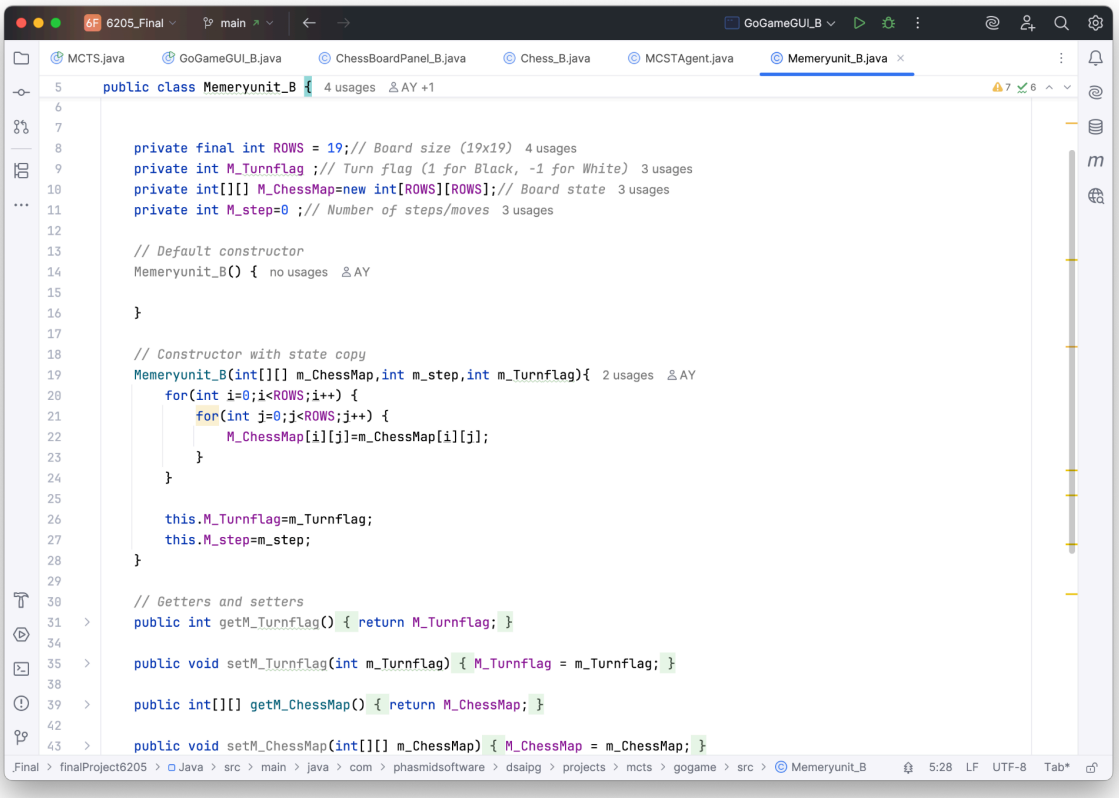


```java
public class MCTS {  AY

    public static void main(String[] args) {  AY
        MCTS mcts = new MCTS(new TicTacToeNode(new TicTacToe().new TicTacToeState()));
        Node<TicTacToe> root = mcts.root;

        int iterations = 1000;

        for (int i = 0; i < iterations; i++) {
            simulate(root);
        }


        for (Node<TicTacToe> child : root.children()) {
            System.out.println("Move:\n" + child.state());
            System.out.printf("Wins: %d, Playouts: %d, Win Rate: %.2f\n",
                    child.wins(), child.playouts(), child.playouts() > 0 ?
                            (double) child.wins() / (2 * child.playouts()) : 0.0);
        }


        Node<TicTacToe> best = root.children().stream() Stream<Node<...>>
                .max(( Node<TicTacToe> a,  Node<TicTacToe> b) -> Double.compare(
                        (double) a.wins() / a.playouts(),
                        (double) b.wins() / b.playouts())) Optional<Node<...>>
                .orElse( other: null);

        if (best != null) {
            System.out.println("Best move chosen:");
            System.out.println(best.state());
        } else {
            System.out.println("No best move found.");
        }
```

(MCTS.java)

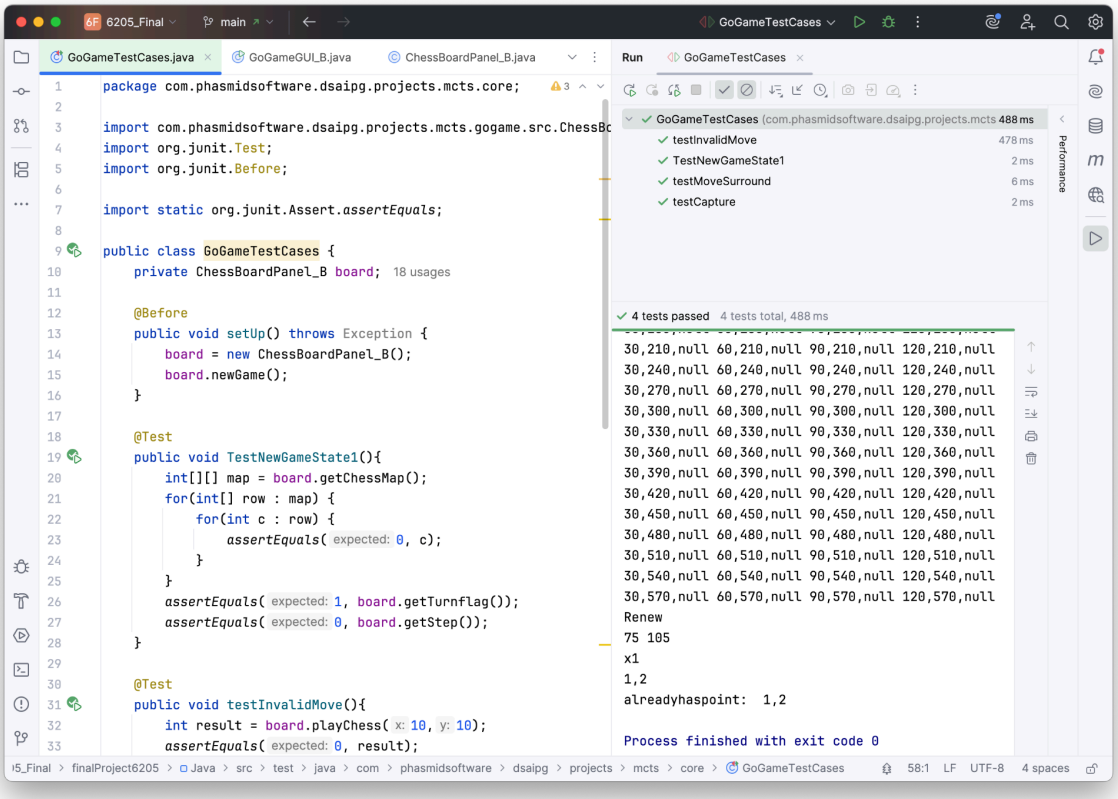


(MCSTAgent.java)



(Memeryunit_B.java)

Test Case Screenshot



(GoGameTestCases.java)