Program Structures and Algorithms

Spring 2025

Final Project Report

NAME: Kaining Yang

NUID: 002054603

GITHUB LINK: https://github.com/yknNEU/INFO_6205_Final_Project

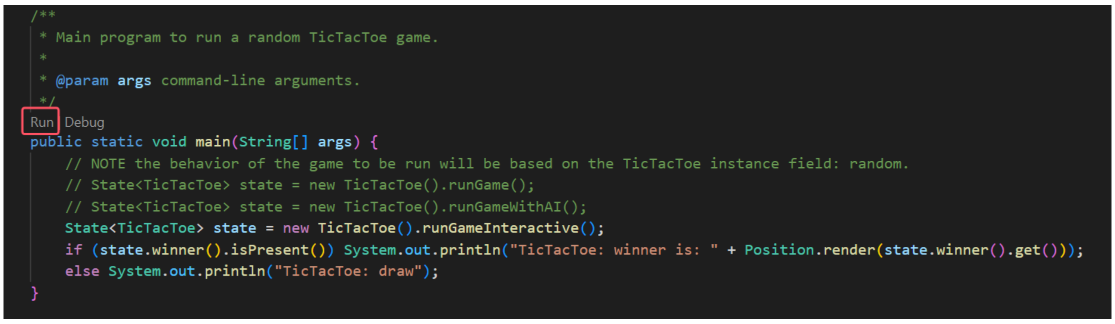
Note: I do not have teammates so this final project is completely my own work.

1. Guidance to run the code from git.

Open your git bash and run the following command:

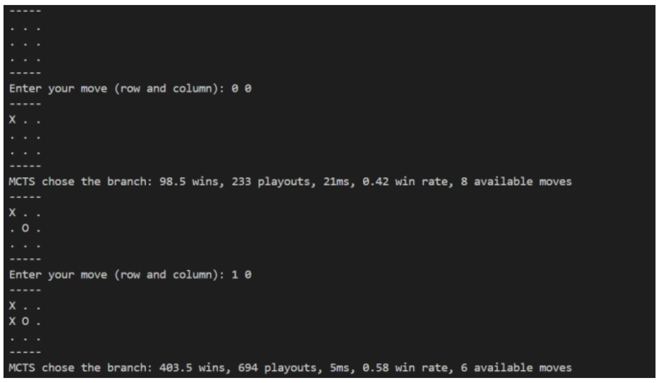
git clone https://github.com/yknNEU/INFO_6205_Final_Project

Then you can see this project in your local directory. Open this folder in any IDE (here I recommend to use VSCode to open the whole folder) and try to run the main function in file *src\main\java\com\phasmidsoftware\dsaipg\projects\mcts\tictactoe\TicTacToe.java* and *src\main\java\com\phasmidsoftware\dsaipg\projects\mcts\chess\Chess.java*. If you are using VSCode and have Java Extension Pack installed, you can see a button named "Run | Debug" above the main function, then you can simply click the "Run" button to run this project.



```java
/**
 * Main program to run a random TicTacToe game.
 *
 * @param args command-line arguments.
 */
public static void main(String[] args) {
    // NOTE the behavior of the game to be run will be based on the TicTacToe instance field: random.
    // State<TicTacToe> state = new TicTacToe().runGame();
    // State<TicTacToe> state = new TicTacToe().runGameWithAI();
    State<TicTacToe> state = new TicTacToe().runGameInteractive();
    if (state.winner().isPresent()) System.out.println("TicTacToe: winner is: " + Position.render(state.winner().get()));
    else System.out.println("TicTacToe: draw");
}
```

By default, you will be the player to play the Tic-Tac-Toe and a Chess game (Go-moku) with an MCTS AI. You need to insert the coordinates where you want to place your piece in your turn. The index of the coordinate begins from 0, which represents the up-left corner of the board. After you have made your move, please be patient to wait a few seconds (about 30 seconds) for the AI to think about its move. After the AI has made its move, it will also show the wins, playouts and time costs of its round.

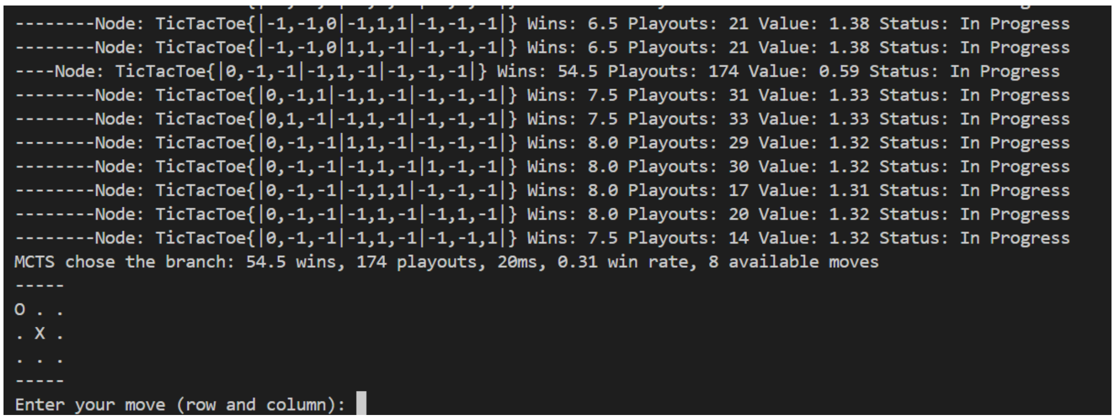


```
-----
. . .
. . .
. . .
-----
Enter your move (row and column): 0 0
-----
X . .
. . .
. . .
-----
MCTS chose the branch: 98.5 wins, 233 playouts, 21ms, 0.42 win rate, 8 available moves
-----
X . .
. O .
. . .
-----
Enter your move (row and column): 1 0
-----
X . .
X O .
. . .
-----
MCTS chose the branch: 403.5 wins, 694 playouts, 5ms, 0.58 win rate, 6 available moves
```

If you want to see how AI plays with itself, you can comment this line:
State<TicTacToe> state = new TicTacToe().runGameInteractive();
and uncomment this line:
// State<TicTacToe> state = new TicTacToe().runGameWithAI();
then run the code again. Or else, if you want to generate a random game, you can uncomment this line instead:
// State<TicTacToe> state = new TicTacToe().runGame();

If you want to modify the search depth of the MCTS algorithm, you can go to the function named *runGameWithAI()* or *runGameInteractive()*, and find this line:
mcts.runIterations(1000); // Run 1000 iterations of MCTS
You can change the number 1000 to any other number that you want.

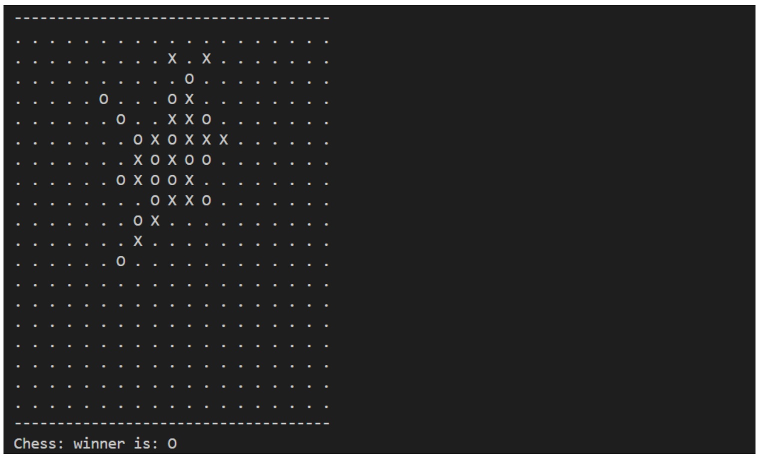
If you want to view the MCTS tree structure, you can uncomment this line:
// mcts.printTreeStructure(null, 0, 1); // debug
(the third parameter represents the depth to print), or this line:
// mcts.printBestPath(root, 2147483647); // debug
then run the code again.



```
--------Node: TicTacToe{|-1,-1,0|-1,1,1|-1,-1,-1|} Wins: 6.5 Playouts: 21 Value: 1.38 Status: In Progress
--------Node: TicTacToe{|-1,-1,0|1,1,-1|-1,-1,-1|} Wins: 6.5 Playouts: 21 Value: 1.38 Status: In Progress
----Node: TicTacToe{|0,-1,-1|-1,1,-1|-1,-1,-1|} Wins: 54.5 Playouts: 174 Value: 0.59 Status: In Progress
--------Node: TicTacToe{|0,-1,1|-1,1,-1|-1,-1,-1|} Wins: 7.5 Playouts: 31 Value: 1.33 Status: In Progress
--------Node: TicTacToe{|0,1,-1|-1,1,-1|-1,-1,-1|} Wins: 7.5 Playouts: 33 Value: 1.33 Status: In Progress
--------Node: TicTacToe{|0,-1,-1|1,1,-1|-1,-1,-1|} Wins: 8.0 Playouts: 29 Value: 1.32 Status: In Progress
--------Node: TicTacToe{|0,-1,-1|-1,1,-1|1,-1,-1|} Wins: 8.0 Playouts: 30 Value: 1.32 Status: In Progress
--------Node: TicTacToe{|0,-1,-1|-1,1,1|-1,-1,-1|} Wins: 8.0 Playouts: 17 Value: 1.31 Status: In Progress
--------Node: TicTacToe{|0,-1,-1|-1,1,-1|-1,1,-1|} Wins: 8.0 Playouts: 20 Value: 1.32 Status: In Progress
--------Node: TicTacToe{|0,-1,-1|-1,1,-1|-1,-1,1|} Wins: 7.5 Playouts: 14 Value: 1.32 Status: In Progress
MCTS chose the branch: 54.5 wins, 174 playouts, 20ms, 0.31 win rate, 8 available moves
-----
O . .
. X .
. . .
-----
Enter your move (row and column):
```

2. **Introduce to Go-moku.**
Go-moku, also known as Five-in-a-Row, is a classic board game that has captivated players for decades. Originating in China, the game has gained international popularity and is known for its simplicity and strategic depth, making it a favorite among both children and adults alike. The game is played on a simple 19x19 grid, and the goal is to be the first player to place five of your stones in a row, either horizontally, vertically, or diagonally.
In Go-moku, the game can be played on a 19x19 grid, and smaller and larger grids can also be used. Each player takes turns placing one stone of their color on an empty intersection of the grid. The game does not have a limit on the number of stones each player can place, and players can strategically place stones to block their opponent's attempts to form a five-in-a-row sequence while working on their own.

The game ends when a player successfully creates a five-in-a-row sequence with their stones, or when all intersections on the board are occupied and no five-in-a-row can be formed by either player. In the latter case, the game is considered a draw.



```
-------------------------------------
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . X . X . . . . . . .
. . . . . . . . . . O . . . . . . . .
. . . . . O . . . O X . . . . . . . .
. . . . . . O . . X X O . . . . . . .
. . . . . . . O X O X X X . . . . . .
. . . . . . . X O X O O . . . . . . .
. . . . . . O X O O X . . . . . . . .
. . . . . . . . O X X O . . . . . . .
. . . . . . . O X . . . . . . . . . .
. . . . . . . X . . . . . . . . . . .
. . . . . . O . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
-------------------------------------
Chess: winner is: O
```

Go-moku is a game of deep strategic thought and foresight. Players must not only consider their next move but also the potential moves their opponent might make in response. This makes the game highly dynamic and engaging, as players must constantly adapt their strategy to the changing board state.

One key strategy in Go-moku is to create multiple potential five-in-a-rows in different directions, forcing your opponent to block one or the other, thus creating an opening for you to complete a line elsewhere on the board.

Another important aspect is to prevent your opponent from creating a five-in-a-row while focusing on your own. This can be achieved by carefully placing stones in positions that block potential lines of five, such as in the middle of a four-in-a-row.

[Note: this introduction is a combination of my own point of view and generative AI]

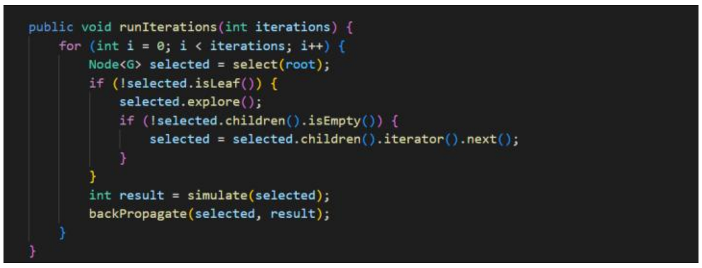
3. **Code Structure**

I have made some minor changes to the skeleton. First, I changed the data type of "*wins*" from int to double in the class "*Node<G extends Game>*", this is because I need to assign 1 to wins, 0 to loses and 0.5 to draws, so int is not sufficient. Second, the wins and playouts are completely handled by the "*MCTS*" class, so in "*Node*" class I just simply initialize them to 0, that's why I have deleted one of the unit tests for the "*Node*" class, and the "*backpropagate*" method is also handled by the "*MCTS*" class but not the "*Node*" class, so I just left the implementation of "*backpropagate()*" method empty in the "*Node*" class.

I also changed the "*MCTS*" class to be generic, so that it can be used for any game as long as I implement the "*Game*" interface. That's why you will not see the "*MCTS*" class in the "*Chess*" package, because I just simply reuse the "*MCTS*" class from the "*TicTacToe*" package.

4. **Introduce to *MCTS***

There are 4 main steps in each iteration of the *MCTS*: select, expand, simulate, and backpropagate. In the "select" step, I will select a candidate node based on its wins and playouts, a node with a higher wins and lower playouts will be more likely to be selected, this is calculated by the UCB. In the "expand" step, if the candidate node has no child, it will be expanded, otherwise the *MCTS* will randomly select one of its children. Then in the "simulate" step, I will randomly play the game until there is a winner or a draw, then I will assign 1.0 wins to the winner, 0.5 to the draw, and 0.0 to the loser. In the "backpropagate" step, I will accumulate the wins and playouts from the child to the root, so if the root has a higher wins, it means that the root is more likely to be a winning state, so in the next iteration, the MCTS will more likely to select this branch.

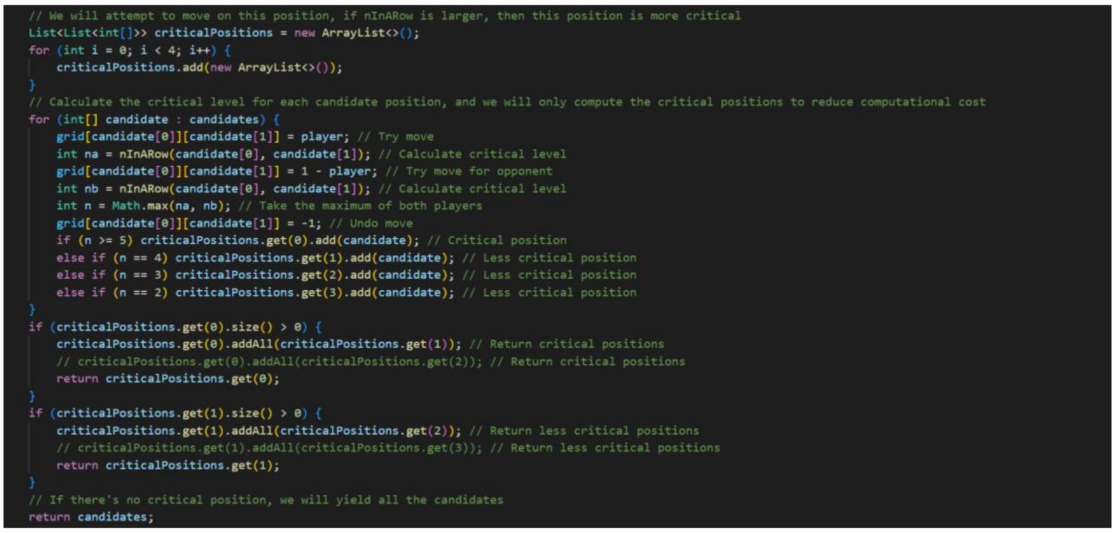


```java
public void runIterations(int iterations) {
    for (int i = 0; i < iterations; i++) {
        Node<G> selected = select(root);
        if (!selected.isLeaf()) {
            selected.explore();
            if (!selected.children().isEmpty()) {
                selected = selected.children().iterator().next();
            }
        }
        int result = simulate(selected);
        backPropagate(selected, result);
    }
}
```

5. **Optimization for the Go-moku game**

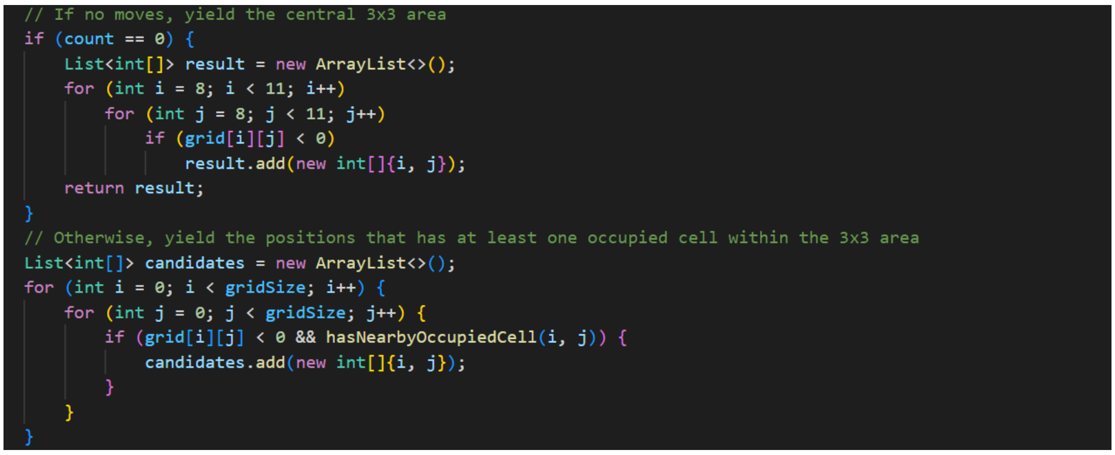
The tic-tac-toe game is a very simple game and the number of possible states is very small, so even without any optimization, the MCTS still spends only about 20ms to run 1000 iterations. But for the Go-moku game, the number of possible states is too large, because the game is played on a 19x19 board, which means in each step, there are over 300 possible moves, without optimization, the MCTS can neither good enough to find the best move, nor can it finish the game within a reasonable time. In fact, I did try to measure how much time it will take to run 1000 iterations in a Go-moku game without any optimization, but I waited for over 5 minutes and it still didn't finish, so I'm not able to show in this report that after optimization, the MCTS runs how many times faster than before. But I can confirm that, after optimization, the MCTS runs at least 100 times faster than before.

The tic-tac-toe game runs fast because it has a very small number of states, so a natural idea to optimize the Go-moku game is to reduce the number of states, in this way, we will able to pure most of the branches in the tree, so that the MCTS can spend more time exploring the promising branches. We can know from the rule of the Go-moku game that, if we find 4 consecutive pieces of the same color, that would mean that this area is critical because if we don't perform any action, the game will immediately end in the next step, so, in this situation, we can force the MCTS to only focus on this area by only yielding this area as possible moves in the next step, this can significantly reduce the number of states during the simulation phase and thus significantly speed up the MCTS. Similarly, the area with 3 consecutive pieces of the same color is a semi-critical area, and the area with 2 consecutive pieces of the same color is a less-critical area. In this way, we can rank these areas based on their criticality and only yield the top k areas as the possible moves in the next step, so that we can not only reduce a great number of states, but also focus on the most promising branches. The way I used to implement this is to try to place a piece at these areas, and calculate there are how many consecutive pieces, and then sort these areas in a descending order by the number of consecutive pieces and restore the placed piece, and only yield the top 2 areas as the possible moves in the next step. So, in fact you might see that "run game randomly" is not totally random, it just places the piece randomly in the top 2 critical areas. But this will not affect human players, you can still choose any valid position to place your piece.

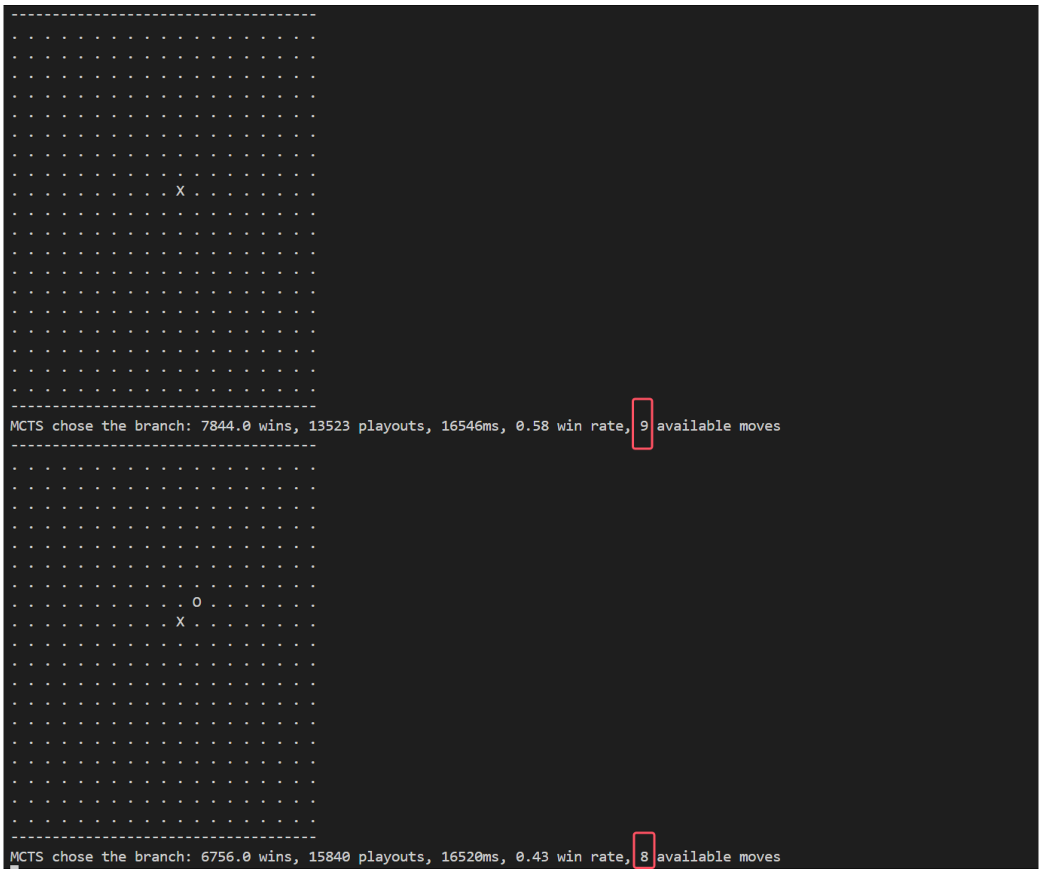


```java
// We will attempt to move on this position, if nInARow is larger, then this position is more critical
List<List<int[]>> criticalPositions = new ArrayList<>();
for (int i = 0; i < 4; i++) {
    criticalPositions.add(new ArrayList<>());
}
// Calculate the critical level for each candidate position, and we will only compute the critical positions to reduce computational cost
for (int[] candidate : candidates) {
    grid[candidate[0]][candidate[1]] = player; // Try move
    int na = nInARow(candidate[0], candidate[1]); // Calculate critical level
    grid[candidate[0]][candidate[1]] = 1 - player; // Try move for opponent
    int nb = nInARow(candidate[0], candidate[1]); // Calculate critical level
    int n = Math.max(na, nb); // Take the maximum of both players
    grid[candidate[0]][candidate[1]] = -1; // Undo move
    if (n >= 5) criticalPositions.get(0).add(candidate); // Critical position
    else if (n == 4) criticalPositions.get(1).add(candidate); // Less critical position
    else if (n == 3) criticalPositions.get(2).add(candidate); // Less critical position
    else if (n == 2) criticalPositions.get(3).add(candidate); // Less critical position
}
if (criticalPositions.get(0).size() > 0) {
    criticalPositions.get(0).addAll(criticalPositions.get(1)); // Return critical positions
    // criticalPositions.get(0).addAll(criticalPositions.get(2)); // Return critical positions
    return criticalPositions.get(0);
}
if (criticalPositions.get(1).size() > 0) {
    criticalPositions.get(1).addAll(criticalPositions.get(2)); // Return less critical positions
    // criticalPositions.get(1).addAll(criticalPositions.get(3)); // Return less critical positions
    return criticalPositions.get(1);
}
// If there's no critical position, we will yield all the candidates
return candidates;
```

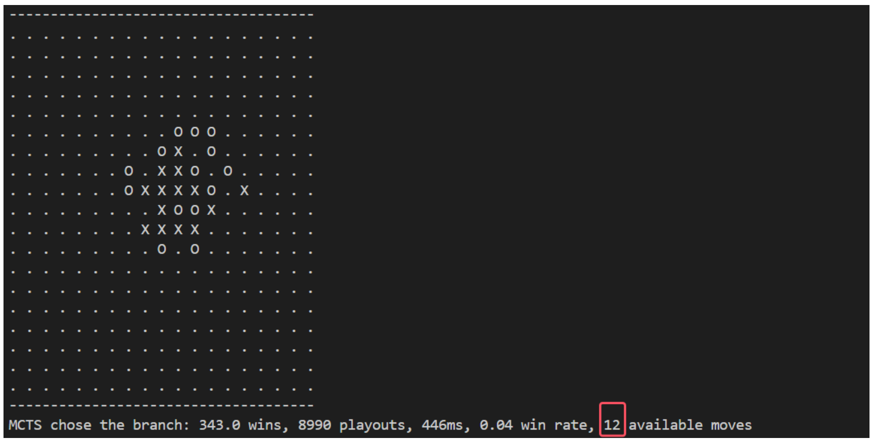
Another idea to optimize the MCTS is that I only allow the MCTS to examine the position that has at least 1 piece in its 3x3 area, this is because during a chess game we need to interact with the opponent, so placing a piece in a completely empty area is rare and can be ignored. And if there are no pieces in the whole board, I will just yield the area in the center of the board as the possible moves. By combining this idea with the previous one, I can reduce the number of possible moves from over 300 to about 20, which significantly speeds up the MCTS. You can see the actual explored moves in the console when running this project as previously mentioned.



```java
// If no moves, yield the central 3x3 area
if (count == 0) {
    List<int[]> result = new ArrayList<>();
    for (int i = 8; i < 11; i++)
        for (int j = 8; j < 11; j++)
            if (grid[i][j] < 0)
                result.add(new int[]{i, j});
    return result;
}
// Otherwise, yield the positions that has at least one occupied cell within the 3x3 area
List<int[]> candidates = new ArrayList<>();
for (int i = 0; i < gridSize; i++) {
    for (int j = 0; j < gridSize; j++) {
        if (grid[i][j] < 0 && hasNearbyOccupiedCell(i, j)) {
            candidates.add(new int[]{i, j});
        }
    }
}
```

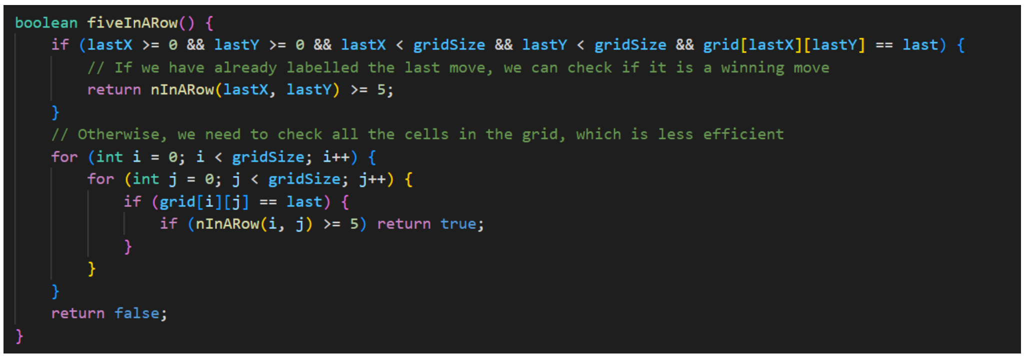


```
-------------------------------------
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . X . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
-------------------------------------
MCTS chose the branch: 7844.0 wins, 13523 playouts, 16546ms, 0.58 win rate, 9 available moves
-------------------------------------
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . O . . . . . . .
. . . . . . . . . . X . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
-------------------------------------
MCTS chose the branch: 6756.0 wins, 15840 playouts, 16520ms, 0.43 win rate, 8 available moves
```



```
-------------------------------------
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . O O O . . . . . .
. . . . . . . . . O X . O . . . . . .
. . . . . . . O . X X O . O . . . . .
. . . . . . . O X X X X O . X . . . .
. . . . . . . . . X O O X . . . . . .
. . . . . . . . X X X X . . . . . . .
. . . . . . . . . O . O . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
-------------------------------------
MCTS chose the branch: 343.0 wins, 8990 playouts, 446ms, 0.04 win rate, 12 available moves
```

Apart from the MCTS, I also did some minor optimizations for the game itself. For example, I cached the position of the last move, so that I can quickly find the winner without need to iterate over the whole board.

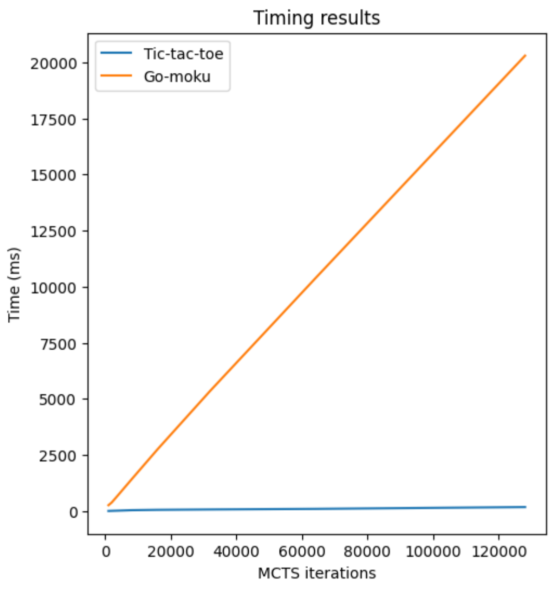


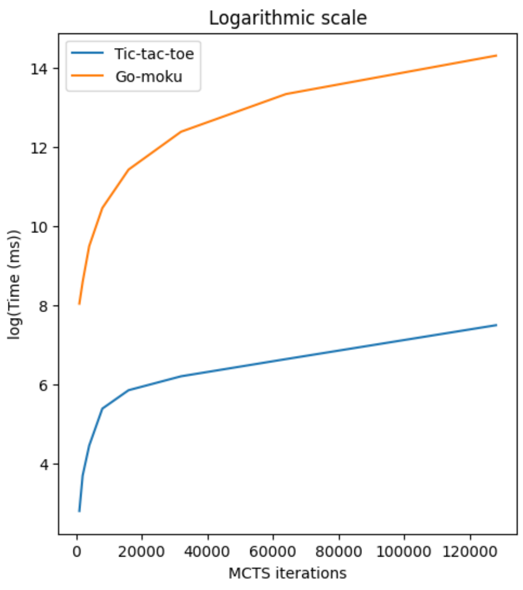
```java
boolean fiveInARow() {
    if (lastX >= 0 && lastY >= 0 && lastX < gridSize && lastY < gridSize && grid[lastX][lastY] == last) {
        // If we have already labelled the last move, we can check if it is a winning move
        return nInARow(lastX, lastY) >= 5;
    }
    // Otherwise, we need to check all the cells in the grid, which is less efficient
    for (int i = 0; i < gridSize; i++) {
        for (int j = 0; j < gridSize; j++) {
            if (grid[i][j] == last) {
                if (nInARow(i, j) >= 5) return true;
            }
        }
    }
    return false;
}
```

6. Timing of runs

As previously mentioned, I cannot measure the times of the Go-moku game before optimization because it really costs too much time. But I measured the time costs under different iterations both on Tic-tac-toe and Go-moku game.

Iterations	Tic-tac-toe	Go-moku
1000	7ms	264ms
2000	13ms	386ms
4000	22ms	723ms
8000	42ms	1408ms
16000	58ms	2761ms
32000	74ms	5357ms
64000	100ms	10363ms
128000	181ms	20295ms





From the graph we can see that the complexity of MCTS is about O(n), where n is the number of iterations.

But in fact, the time spent is very unstable, especially for the optimized Go-moku game. That is because, in the Go-moku game, I only yield the most critical positions, so in some status, there are no critical positions and I have to return all the possible moves, in this situation it will take much more time, in some other status, there are only a few critical positions, especially for the status that the game will end in a few steps, in these cases the MCTS can finish quickly. We can also see the same phenomenon in the tic-tac-toe game, that the MCTS runs faster and faster, because the closer we are to the end of the game, the less available status we have.

As for the performance, 1000 iterations are enough for the MCTS not to lose any tic-tac-toe game, and I cannot win the Go-moku game with 100000 iterations of the MCTS.

Conclusion

Monte Carlo Tree Search is a very powerful and flexible algorithm for game-playing. It is very smart with a large number of iterations for decision-making in games. It has a linear complexity. But it still has some limitations, if there are too many available choices, it may need to spend too much time to find the best choice among them, besides, it is computational expensive, we need to perform some optimization strategy to pure some branch to reduce its complexity, so that it can work more efficiently.