INFO 6205 Spring 2025 Team Project

Group 2

Wenbo Jin, Jiacheng Cui, Ziyuan Yin

Summary

In this project, our team implements Monte Carlo Tree Search (MCTS) for TicTacToe and a Merge game similar to the game named 2048. We focused on developing MCTS frameworks, applying them to both games, and analyzing the behavior through extensive experiments. In particular, the Merge game is the main target for optimization in our project, with iteration-based comparisons and performance evaluation.

1 Introduction

Monte Carlo Tree Search (MCTS) is a powerful approach to designing game-playing bots or solving sequential decision problems. The method relies on intelligent tree search that balances exploration and exploitation.[1]. It builds a search tree incrementally and balances exploration and exploitation by performing randomized simulations. MCTS typically consists of four key steps:

1. Selection: Starting from the root, the algorithm selects child nodes based on a policy called Upper Confidence Bound (UCB) until it reaches a leaf node.
2. Expansion: If the selected node is not terminal, one or more child nodes are added to expand the tree.
3. Simulation: A simulation is run from the newly expanded node to a terminal state using a selected policy (randomly or other specific policy).
4. Backpropagation: The result of the simulation is propagated back up the tree to update the statistics of the visited nodes.

By repeating those processes many times, MCTS gradually improves its performance. Then it can give relatively promising ‘moves’ for games with decision spaces.

In this project, we applied the Monte Carlo Tree Search algorithm to two classic games, Tic Tac Toe and 2048 (merge game), and analyzed the simulation results.

2 Implementation and Simulation

2.1 MCTS Framework

We followed the framework given in the course material to implement the MCTS framework. The framework consists of four core interfaces: ‘Game’, ‘State’, ‘Move’ and ‘Node’. These abstraction classes allow us to apply MCTS to different games without rewriting the core structure.

2.2 TicTacToe Simulation

We implemented MCTS for the TicTacToe game, following the course material structure. The main part of this game is implemented in ‘TicTacToe.java’ and game simulation is implemented in ‘MCTS.java’. Our implementation with UI is shown in the figure below.

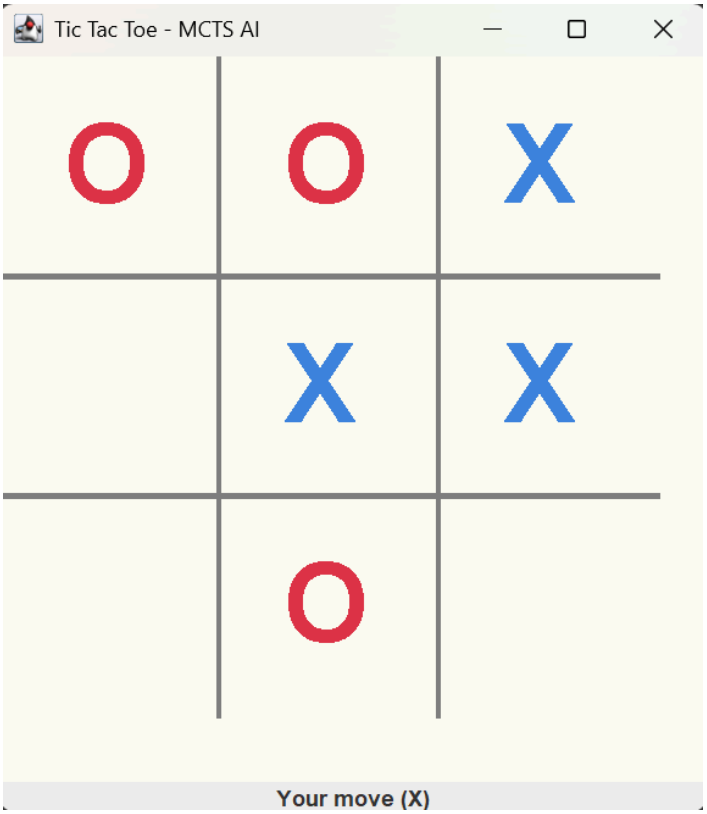

Figure 2.1 TicTacToe Implementation

All tests for TicTacToe have passed and the screenshot is attached in the appendix.

2.3 Merge Game Simulation

Our team chose a popular game named 2048 as our own algorithm simulation. It’s a single-player puzzle where players use directional keys to move blocks as a whole and combine blocks of equal value for fusion. The goal is to achieve the highest possible score by merging larger and larger tiles. Each merge can increase scores and the board complexity. The game ends when no more moves are available which means there is no empty space available and you can’t merge any tiles. When the game ends, you can’t do any more moves and the final score will display on the bottom of the Ul. Our implementation with UI is shown in the figure below.

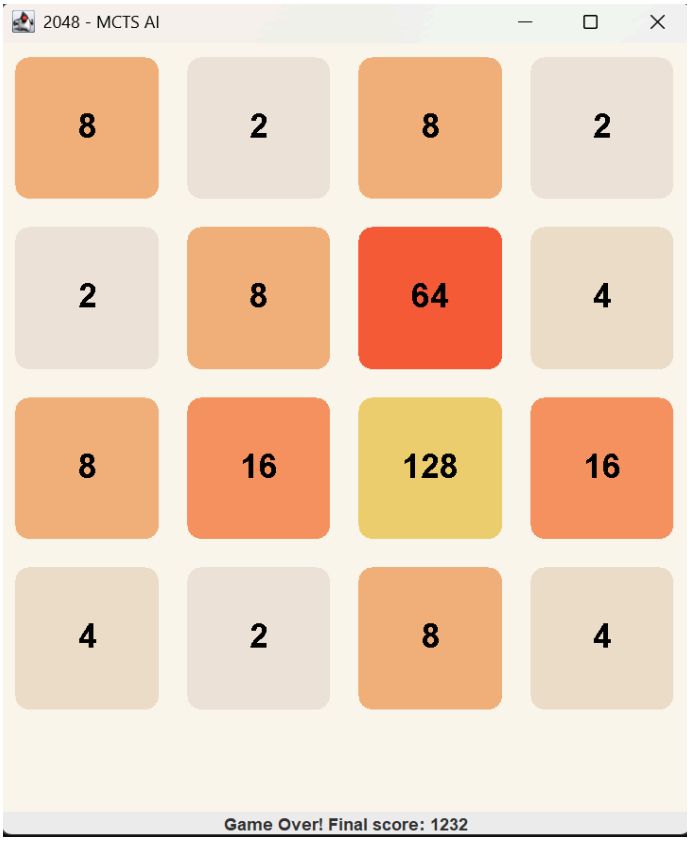

Figure 2.2 Merge Game Implementation

The key class of MergeGame simulation is 'MergeState' and 'MergeNode'. 'MergeState' controls the state of the game board. It provides essential interfaces for MCTS to interact with the game. 'MergeNode' defines the structure of an MCTS and implements four phases of the algorithm. It's also responsible for evaluating states using a customizable utility function. 'MergeSimulator' is the main simulation runner for getting benchmark data.

In order to better analyze the performance of MCTS, we set up multiple sets of iterations for comparison, recording the time spent on each run and the score it got. The data we got from left to right are: iterations,timeMillis,score. To get the data we should run the file named: MergeSimulator.java.

```
1    50,56,4
2    50,25,4
3    50,19,4
4    50,16,4
5    50,15,8
```

Figure 2.3 Running Result of MergeSimulator.javat

In this game, MCTS can give a promising planning strategy for this game due to its sequential decision feature. However, to get better performance and higher score, it requires customized rollouts to effectively explore rewarding tile combinations.

All tests for MergeGame have passed and the screenshot is attached in the appendix.

3 Comparison and Analysis

3.1 TicTacToe Analysis

Instead of choosing a random move, we implemented a check process: first, check if the opponent has a winning move and block it; next, check if the current player can win immediately and take that move; if neither condition is met, fall back to a random valid move. This change shows better results.

3.2 Merge Game Analysis

To Make the MCTS achieve the best performance quickly with the smallest iteration, we can use the logTile Heuristic Strategy.

This enhancement introduced a heuristic utility that rewards states based on the logarithmic value of the max tile. The formula used was:

$$Utility = scoreDelta + 15 \times emptyCells + 10 \times \log_2(maxTile)$$

The results shown like below:

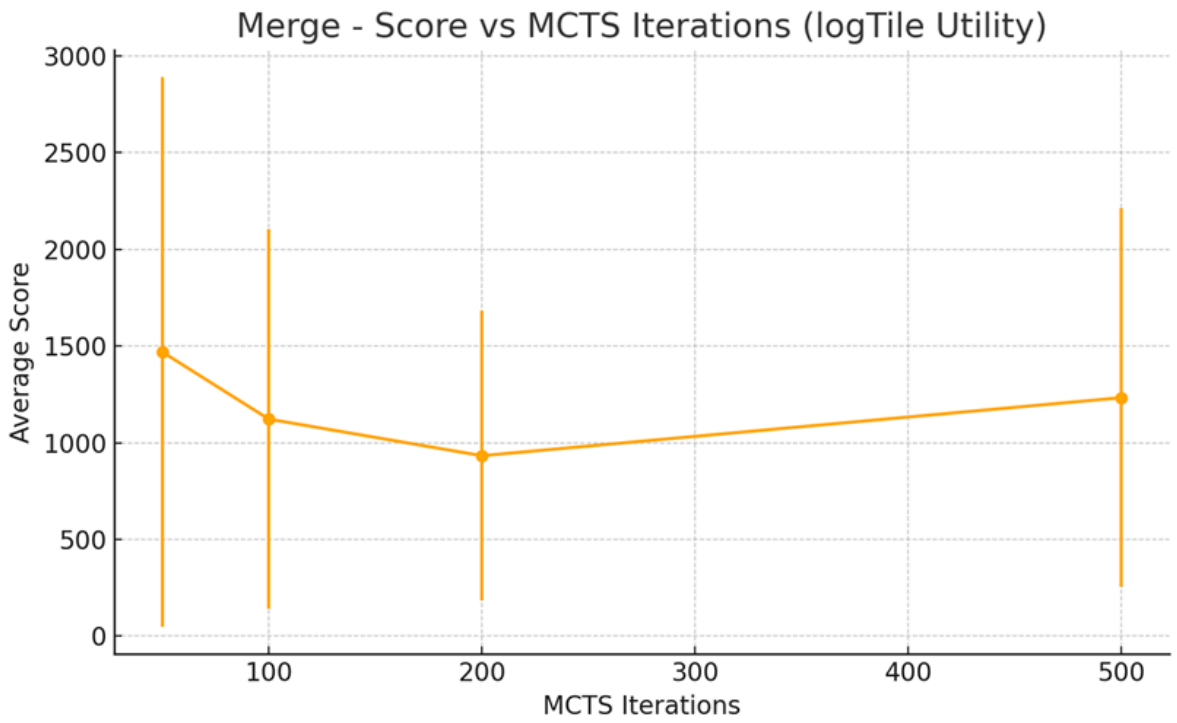

Figure 3.1 Score vs Iterations (logTile Improvement)

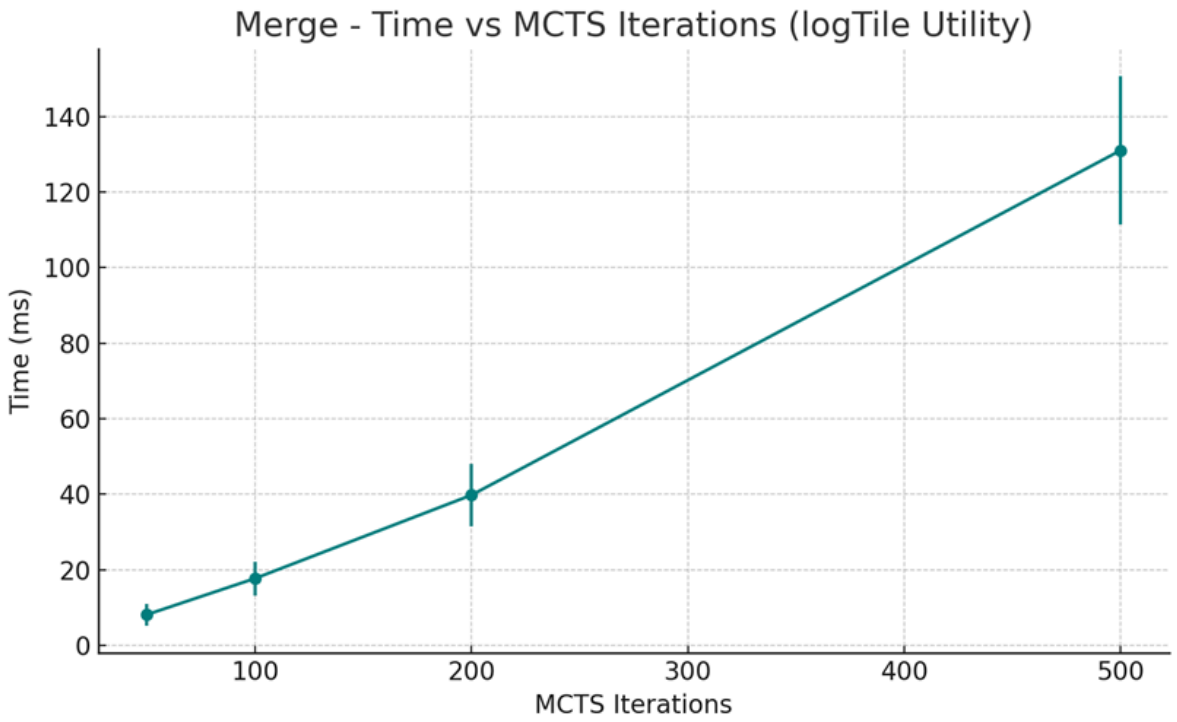

Figure 3.2 Time vs Iterations (logTile Improvement)

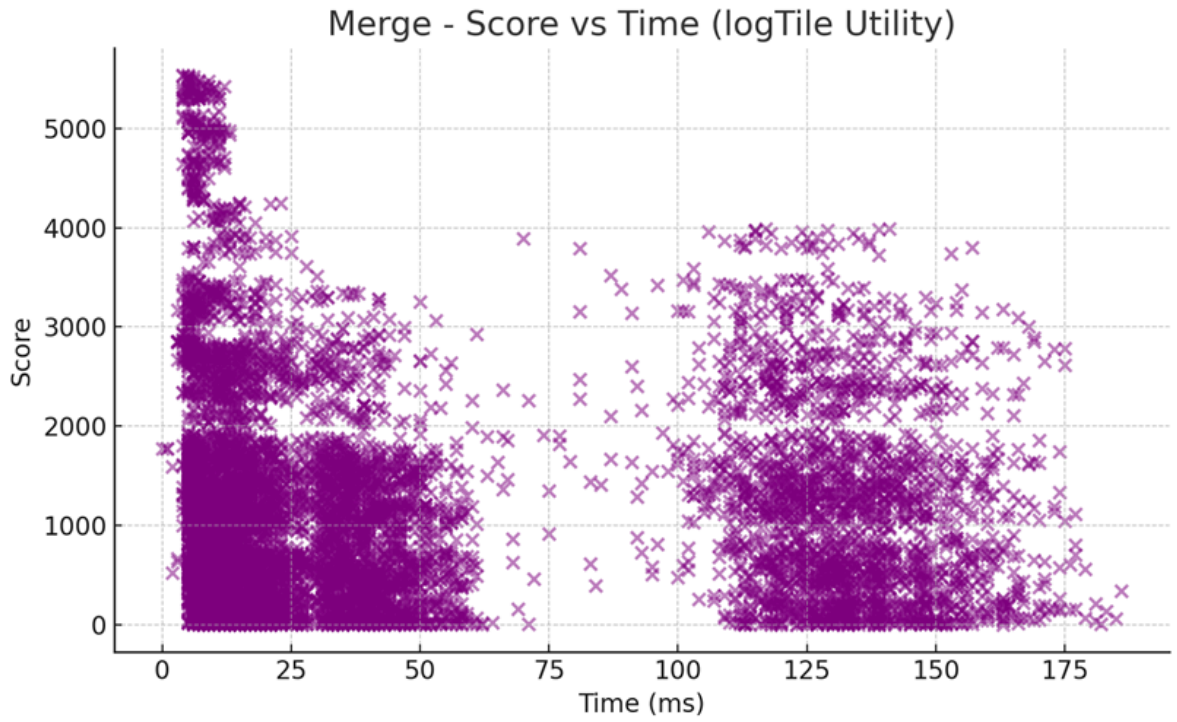

Figure 3.3 Score vs Time (logTile Improvement)

From the graphs we can see that: the best average score is approaching 1500, achieved at 50 iterations with minimal cost and the score vs time graph shows that there are multiple score clusters between 20–100ms. It reaches our goal to make improvements. The log-scale provides better scaling for the exponential feature of the 2048 game. It shows a high average

score at 50 and 500 iterations, suggesting better early guidance and improved deep selection stability.

However, if we want a good performance in deeper simulation, we can let the MCTS strategy become a refined combination of the logTile-inspired utility function and multi-rollout averaging. This setup is trying to preserve the early-game intelligence of logTile while significantly improving simulation stability during mid-to-high iteration counts.

In this method, we can do a little change to the utility formula we mentioned before:

$$Utility = scoreDelta + 15 \times emptyCells + 5 \times \log_2(maxTile)$$

Also, we changed simulation to 2, which means each move is simulated twice and averaged. The rollout depth limit is 20 steps maximum. What's more, when we do move selection, we let it choose the highest average utility across simulations. The results shown like below:

Figure 3.4 Score vs Iterations (Further optimization)

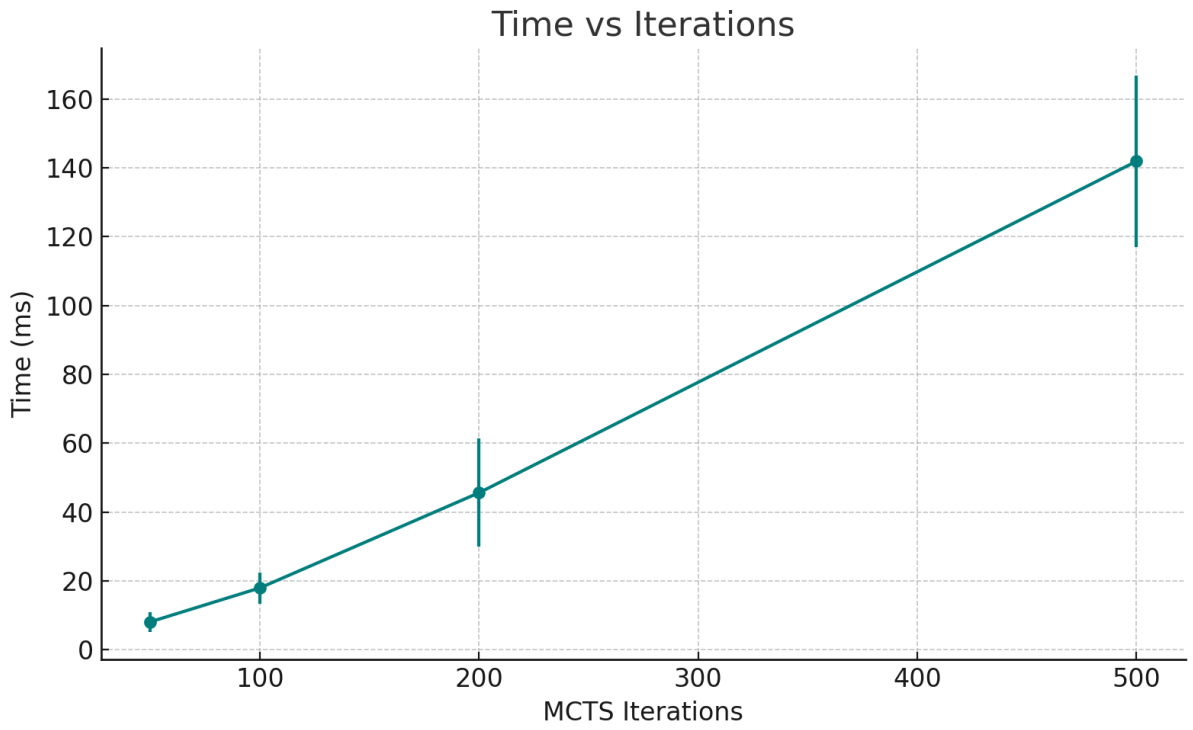

Figure 3.5 Time vs Iterations (Further optimization)

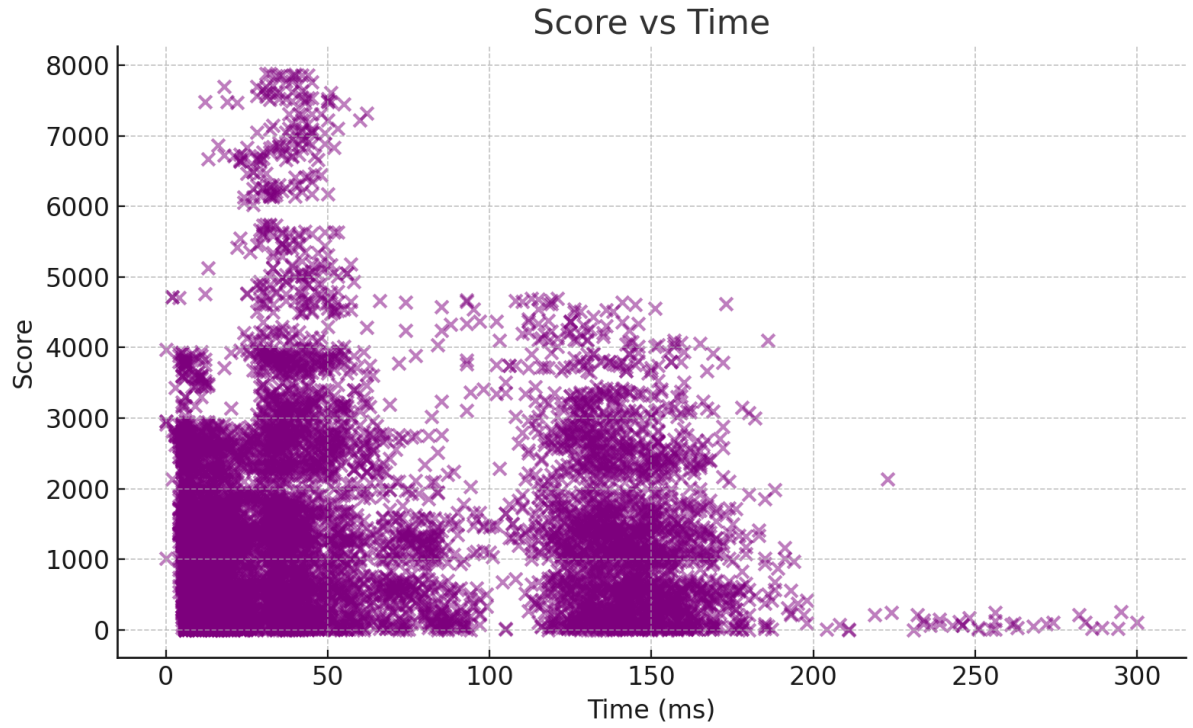

Figure 3.6 Score vs Time (Further optimization)

Compared to the original logTile version, this strategy maintains the same utility structure but doubles the number of simulations per move, making it more resistant to noise. As a result, it achieves much higher average scores at 200 and 500 iterations, peaking around 1950. The results shown below:

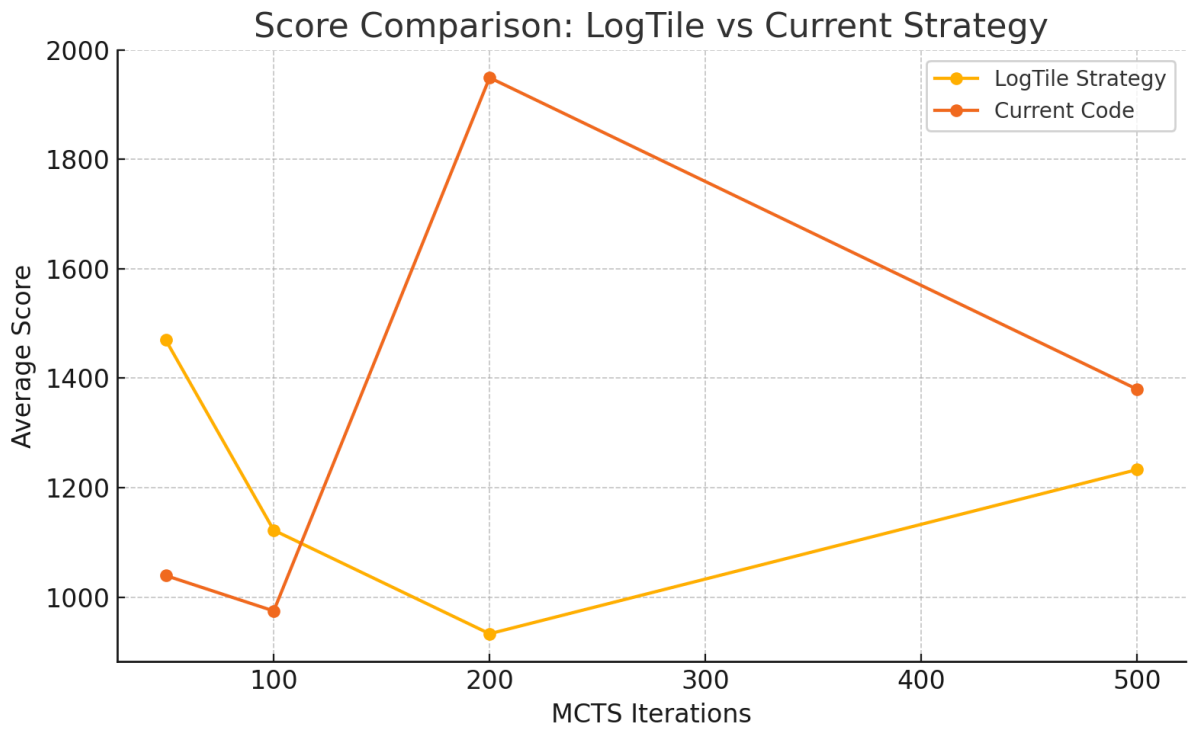

Figure 3.7 Score Comparison of two improvement methods

Also, we compared the runtime efficiency of both the LogTile strategy and the current improvement method across four iteration levels: 50, 100, 200, and 500. From the figure below we can find that the execution time increases linearly with the number of MCTS iterations and both strategies share a similar runtime scaling which means rollout depth and logic are well-controlled. The results shown below:

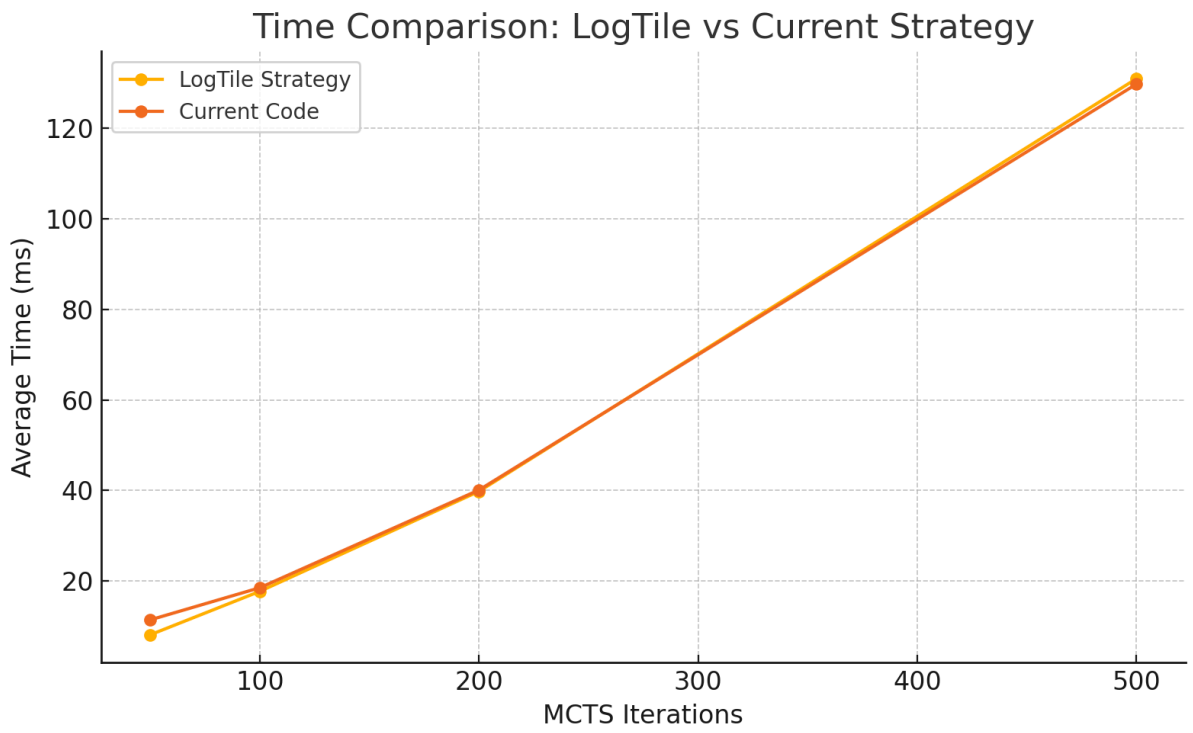

Figure 3.8 Time Comparison of two improvement methods

Overall, both strategies demonstrate highly efficient performance profiles. The minimal time overhead between them underscores that score differences are largely due to the quality of the rollout strategy rather than computational cost. The first improvement can be used when we need good performance at a low iteration level, the second one is very useful when it comes to a higher iteration level.

Reference

[1] M. Świechowski, K. Godlewski, B. Sawicki, and J. Mańdziuk, “Monte Carlo Tree Search: A Review of Recent Modifications and Applications,” *Artificial Intelligence Review*, 2023. [Online]. Available: https://arxiv.org/abs/2103.04931

Appendix: Unit Tests Screenshot

The unit tests are all in path: \projects\src\test. The setting of unit tests has high coverage.

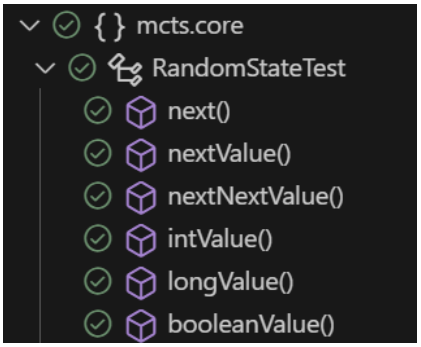

Figure 1 Unit tests for core part

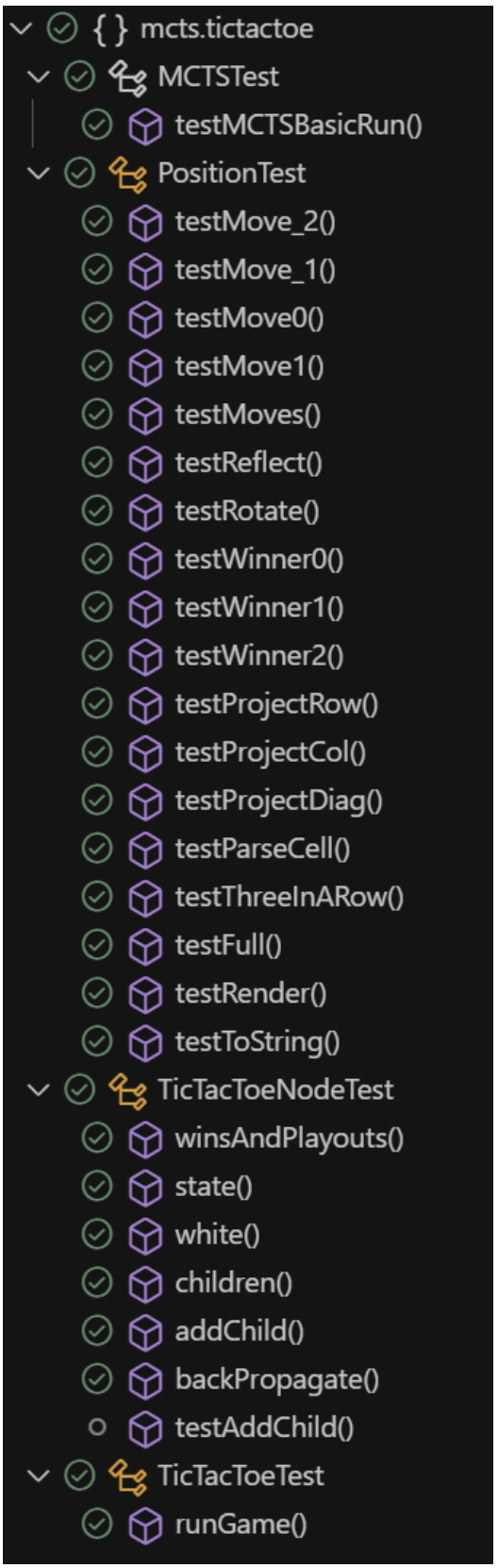

Figure 2 Unit tests for TicTacToe

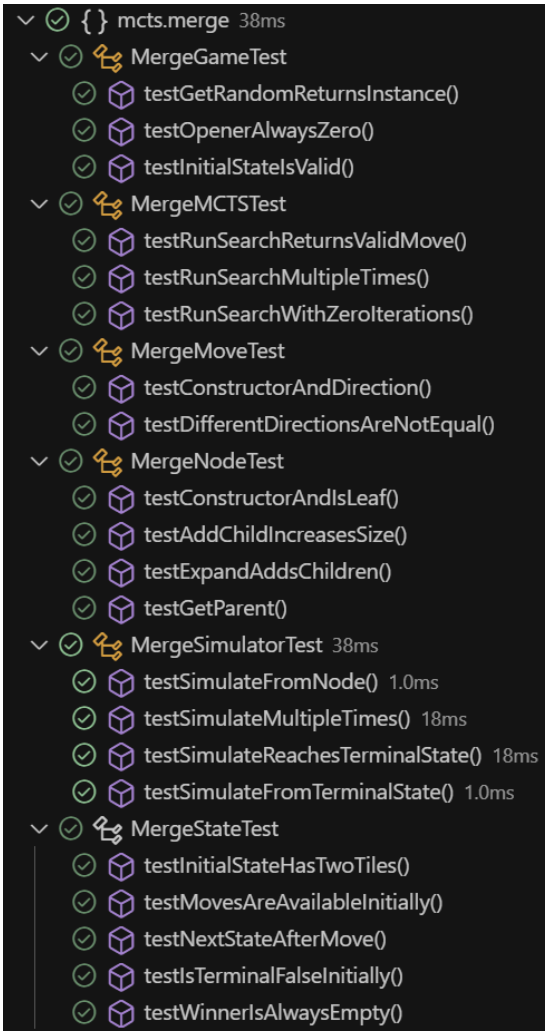

Figure 3 Unit tests for merge game