Tic-Tac-Toe & Nim Game

Shaofan Wei, Renzhou Yuan, Zheng Zhang

[1]Information Systems, Northeastern University
[2]Software Engineering Systems, Northeastern University

wei.shao@northeastern.edu, yuan.re@northeastern.edu, zhang.zheng7@northeastern.edu

***Abstract* – This final project, we are focusing on creating a Nim game by using Monte Carlo Tree Search. In this report, we will discuss how we came up with this idea to build a Nim game. What are the problems that we were facing while developing this game. In this report, we will talk about what problems we faced when we started to look at this project. We will discuss about the MCTS (Monte Carlo Tree Seach) – The key components throughout the whole project. We will talk about the Tic-Tac-Toe game. This game is a sample for the MCTS, this game also gave us a basic guideline about how to build our own game – Nim game. We also did a comparison between the MCTS and Random search algorithm to see how much MCTS has improved the AI to perform a more challenging game experience.**

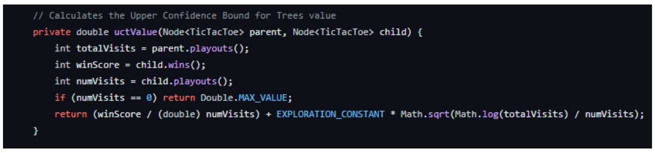
I. PROBLEM DESCRIPTION

In this project, we are required to build our own game by using MCTS (Monte Carlo Tree Search). It is a search algorithm used for making decisions in some types of games. MCTS contains 4 steps: Selection, Expansion, Simulation and Backpropagation. In Selection stage, the algorithm will start at the root node, travel through the node tree based on the selection policy until it reaches the next node that has not been explored; Expansion means that there are more child nodes added to expand the tree; A Simulation means that algorithm plays the game from the current node to the end with all the possible ways and record all those results. After simulation, it selects the best solution; Backpropagation refers to updating the information stored in the nodes it took to reach the terminal node. In order to let the search algorithm work, these 4 essential steps are the main target we should be acknowledge and understand.

II. ANALYSIS

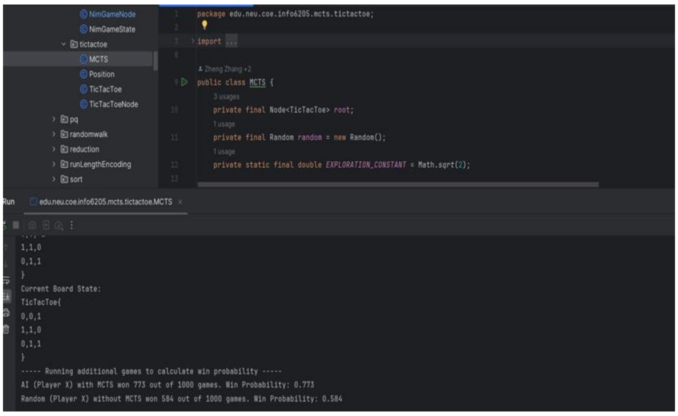
Before building our own game, we first started a sample game Tic-Tac-Toe. The purpose of this game is to help us understand how MCTS has been used in such a game. In the Tic-Tac-Toe, we worked on the TO BE IMPLEMENTED part, Position.java is a detailed implementation of managing the state of the Tic-Tac-Toe game. It defined the game board is a 3 * 3 matrix of grid, -1 represents the grid is empty, 0 represents the grid holds the value ‘O’, 1 represents the grid holds the value ‘X’. The moves() method generates the possible moves for the specified player while move() execute a move by the specified player. The threeInARow() checks all the rows, columns, and diagonals.

The MCTS.java class, is an implementation of MCTS algorithm for the Tic-Tac-Toe game, the class is structured to explore possible game states for the game of Tic-Tac-Toe. The runMCTS (int iterations) is mainly working on running the MCTS algorithm. The select(Node<TicTacToe>) select a node to explore using the UCT (Upper Confidence Bound applied to Trees) formula if all child nodes are fully expanded. If not, it picks a non-fully expanded node for further exploration. The UCT is used in selection phase to balance the exploration of less visited nodes and exploitation of nodes known in order to return good outcomes. In the process of learning MCTS, we found that the Tic-Tac-Toe game does not use heuristic functions.

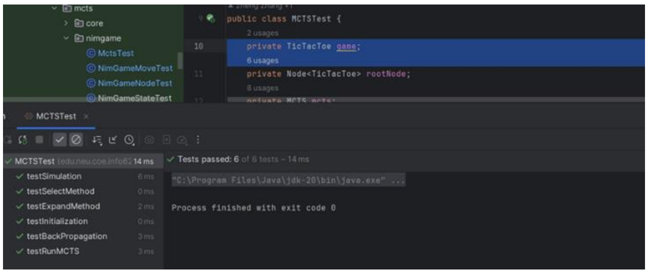


Tic-Tac-Toe:

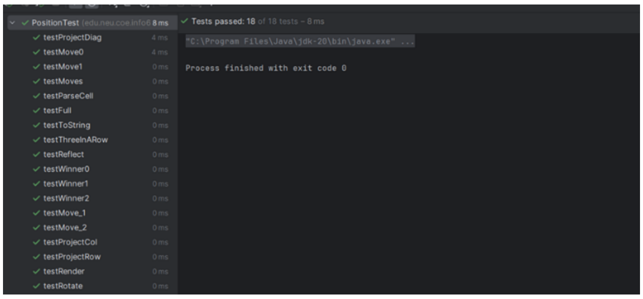
MCTS:



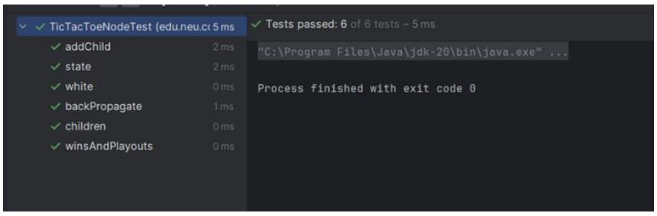
MCTS Test:



Position:



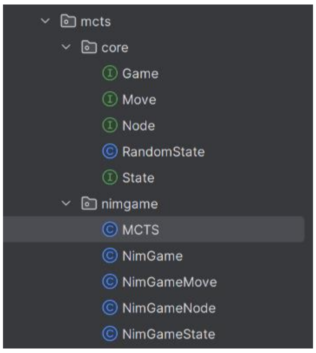
TicTacToeNode:



TicTacToeTest:

III. Implementation

While we were developing our own game, we thought about developing a Nim game. The rule is, there will be 3 pile states once user start this game: pile 1, pile 2 and pile 3, pile 1 contains 3 piles, pile 2 holds 6 piles, pile 3 have 9 piles. The user needs to choose which pile they want to move and enter the number of piles they want to take away. For example, if player entered 1, that means the user wants to move piles at pile 1, and then enter 2 means the user will take 2 piles away, which means there will be 1 pile left at pile 1.



In the MCTS.java that we implemented specifically for Nim game, has some key strategies we used – The heuristic functions particularly used in simulation and expansion phases for the MCTS.

evaluateState():

```
private int evaluateState(NimGameState state) {
    int[] piles = state.getPiles();
    int xorSum = 0;

    for (int pile : piles) {
        xorSum ^= pile;
    }

    return xorSum;
}
```

The function is used to help select the best move during expansion phase. The heuristic applied here involves using the XOR sum of the pile sizes. The XOR of all pile sizes is to determine if a position is winning or losing. If the sum is not zero, then this position is theoretically winning.

selectBestMove():

```
no usages
private Move<NimGame> selectBestMove(Node<NimGame> node, List<Move<NimGame>> moves) {
    int bestScore = Integer.MAX_VALUE;
    Move<NimGame> bestMove = null;

    for (Move<NimGame> move : moves) {
        // Apply the move and get the new state
        State<NimGame> newState = node.state().next(move);

        int score = evaluateState((NimGameState)newState);

        if (score < bestScore) {
            bestScore = score;
            bestMove = move;
        }
    }

    return bestMove;
}
```

The method uses evaluateState() to choose the best moves.

Simulation phase:

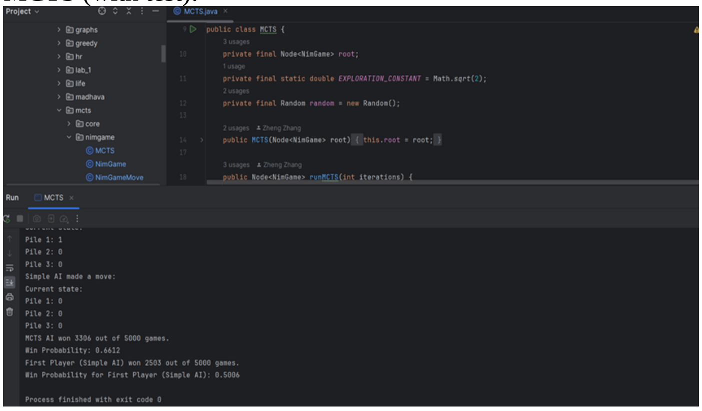
```
1 usage
private Move<NimGame> selectHeuristicMove(List<Move<NimGame>> moves, Node<NimGame> node) {
    // Prioritize moves that reduce the number of stones in the pile the most
    moves.sort((m1, m2) -> Integer.compare(((NimGameMove)m2).getPileReduction(), ((NimGameMove)m1).getPileReduction()));
    return moves.get(0);
}

1 usage
private List<Move<NimGame>> heuristicMoves(Node<NimGame> node) {
    List<Move<NimGame>> moves = new ArrayList<>((List<Move<NimGame>>) node.state().moves(node.state().player()));
    Collections.shuffle(moves);
    return moves;
}
```

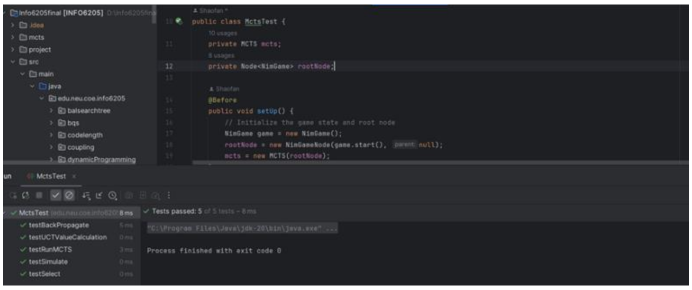
During the Simulation phase, the algorithm uses a simple heuristic to prioritize moves in order to reduce opponent’s options.

IV. Evaluation

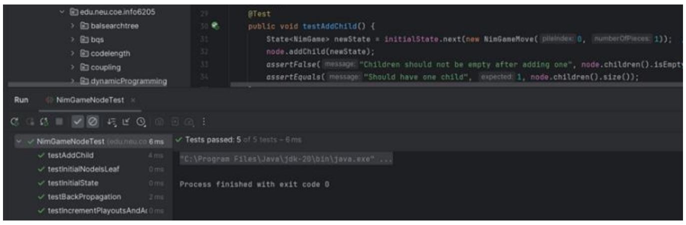
MCTS:
With those implementations, we ran our game to check if the game could work properly.
The screenshot below shows how the Nim game looks like, the pile states have been correctly generated, and the game logic works properly. Other parts of the program also have Unit tests to test whether the methods are running properly.

MCTS (with test):



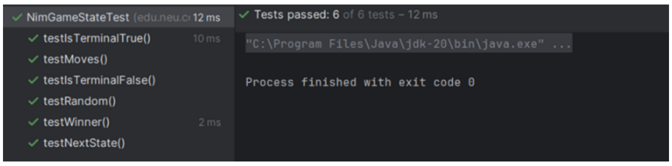


NodeTest:

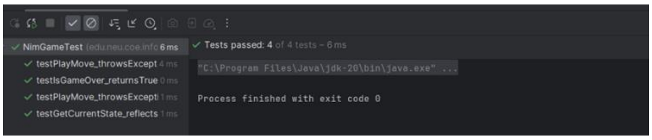


MoveTest:

StateTest:



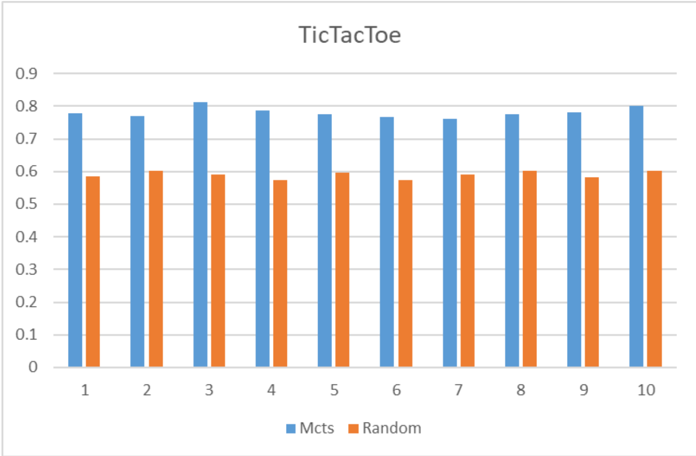
NimGameTest:

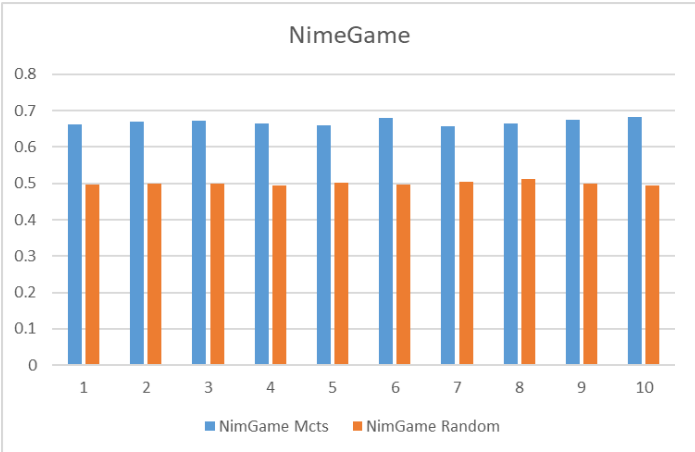


V. Discussion (Reflection)

In order to ensure that MCTS has improved the selection of AI. We compare the running time of the two games with and without MCTS.

The Pictures below are the overviews of the Tic-Tac-Toe/ Nim game with MCTS and without MCTS.





The X-axis represents the test number, the y-axis represents the win rate. For the Tic-Tac-Toe game, the average win rate without MCTS is nearly 0.6, while MCTS could improve the win rate to nearly 0.7.
For the Nim game, the average win rate without MCTS is approximately 0.5. With MCTS, however, the win rate could be approximately 0.7, which is a 0.2 increase.
In conclusion, MCTS does perform more strategic performance, which could make the game more challenging.

VI. Conclusions

In this project, we acknowledge that MCTS is a search algorithm used for decision-making games, it could perform a complex decision-making process through Selection, Expansion, Simulation and Backpropagation. First, we learned what MCTS is by working through the sample Tic-Tac-Toe game. We also dived deeper into the MCTS by developing our own Nim game, which also required heuristic functionality by using XOR sum to determine better moves. We also figured out how MCTS actually impacts the games by comparing the game algorithm with MCTS and Random Search. The results maintained that MCTS does improve the AI to perform better strategies, thus improving the game experiences.

REFERENCES

Jiang, W. (2024). Application of Monte Carlo Tree Search algorithm in Go playing. *Applied and Computational Engineering*, *53*, 274–279. https://doi.org/10.54254/2755-2721/53/20241503

Kemmerling, M., Lütticke, D., & Schmitt, R. H. (2023). Beyond Games: A Systematic Review of Neural Monte Carlo Tree Search Applications. *ArXiv (Cornell University)*, *54*. https://doi.org/10.1007/s10489-023-05240-w

Roy, R. (2019, January 14). *ML | Monte Carlo Tree Search (MCTS)*. GeeksforGeeks. https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/

Wikipedia Contributors. (2019, March 14). *Monte Carlo tree search*. Wikipedia; Wikimedia Foundation. https://en.wikipedia.org/wiki/Monte_Carlo_tree_search