Project 2 - Investigation of the Monte Carlo Tree Search (MCTS) and Minimax Algorithm in a (Combinatorial) Tic-Tac-Toe Game Environment

Shaivan Bhagat

SCICOMP302 Algorithms and Data Structures

This is all my own work. I have not knowingly allowed others to copy my work. This work has not been submitted for assessment in any other context.

1. Description of the Algorithms and Game Environment

1.1 Monte Carlo Tree Search

The Monte Carlo Tree Search (MCTS) is a heuristic search algorithm mainly used for processes involving decision making specifically for software used in board games. It uses the classic tree search algorithm whilst implementing machine learning making its nature probabilistic.

MCTS employs a strategic method of navigating a tree structure by balancing between exploration and exploitation. By conducting random simulations and storing action statistics, it enables more informed decision-making in subsequent iterations. This technique delves only a few layers into the tree and prioritizes specific branches for exploration. Rather than exhaustively expanding the search space, it simulates potential outcomes, reducing the number of evaluations needed. The process involves an evaluation through play-out or simulation, where the algorithm progresses from a starting point to a leaf state by making random decisions. The results are then used to update nodes along the path to the root, ultimately selecting the state with the most favourable rollout score after the simulation.

1.2 *Game Tree*

MCTS is used in combinatorial games, i.e., “sequential games with perfect information”. Tic-tac-toe is a classic example of such a combinatorial game since it has a set of preconditions that it matches:

- 2-player game
- Sequential (players take turns)
- Finite set of defined moves

Due to this game being combinatorial and simple to understand due to its popularity, it was chosen to understand the working of the MCTS. A game tree for tic-tac-toe represents all possible outcomes where a directed graph depicts various game states through its nodes, while the edges symbolize potential subsequent states from each specific state. The terminal nodes

in this graph signify outcomes such as a win, loss, or a draw in the game. Such a game tree is used to measure the complexity of the game.

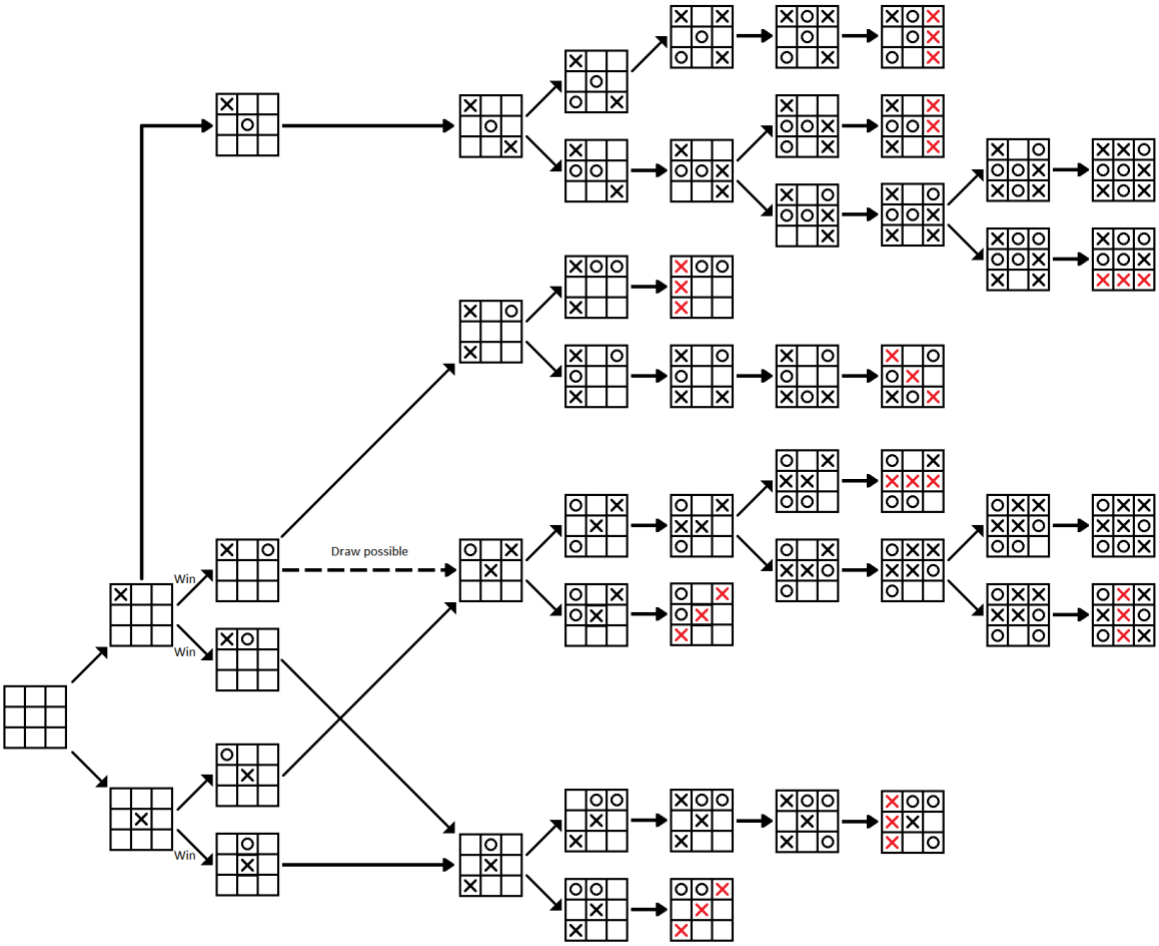

Figure 1: Game tree for a combinatorial Tic-Tac-Toe game

If playing against artificial intelligence (the computer in this case), ideally a move which leads to the winning lead node path must be chosen to win. This can be implemented using the algorithms 1) Minimax, 2) Minimax with alpha, beta pruning, and 3) A* search. Here, I only focus on the Minimax algorithm.

1.3 Minimax Algorithm

The Minimax algorithm is a decision-making approach used in game and decision theory designed for 2-player games. It helps a player (in this case, AI) to determine the best move to make. It works by evaluating the game state and possible moves. The "max" part refers to maximizing the player's advantage, while the "min" part reflects minimizing the opponent's

advantage. The algorithm explores the game tree by simulating potential moves, alternating between maximizing the player's advantage and minimizing the opponent's advantage at each level of the tree. In Tic Tac Toe, the Minimax algorithm is applied to determine the best possible move for a player at any given state of the game. It works by employing the following:

Evaluation: The algorithm evaluates the current state of the game and creates a game tree representing all possible moves for both players until a terminal state (win, lose, or draw) is reached.

Depth-limited search: As Tic Tac Toe has a manageable number of possible moves, the algorithm can explore the entire game tree. However, in more complex games, depth limitation is crucial to keep the computation feasible.

Maximizing and minimizing: Assuming the computer plays 'X' and the player plays 'O', the algorithm alternates between maximizing the 'X' player's advantage and minimizing the 'O' player's advantage at each level of the game tree.

Recursive evaluation: The algorithm recursively evaluates possible moves by assigning scores to each move. For example, a win might be scored as +1, a loss as -1, and a draw as 0.

Choosing the best move: The 'X' player aims to maximize its score (get the highest score), while the 'O' player aims to minimize it (get the lowest score). The algorithm chooses the move that leads to the highest score for 'X' or the lowest score for 'O' at each step.

Optimal move determination: At the top level of the tree, the algorithm selects the move that results in the best outcome for 'X'. This move is considered the optimal move to make.

1.4 Reason for Choice

Given the simplicity of the game Tic-tac-toe, exploring the role on MCTS and Minimax might seem unconventional. However, it serves as decent testing grounds to understand and experiment with the algorithm allowing for quick iterations. It also allows for experimenting and understanding performance. Here, these algorithms are not used to solve the game with a target of a win in every scenario, but to test and optimize the algorithm in a controlled setting and understand how it might be used in more complex problems.

2. Experimental Platform

The algorithms were run on a MacBook Air running on a M2 Chip. General specifications of the hardware are:

- Total Number of Cores: 8 (4 performance & 4 efficiency)
- Memory: 8 GB

- Chipset Model: Apple M2
- Type: CPU
- Bus: Built-In

The algorithms were run completely through Python. Python files were run via Spyder, accessed through Anaconda-Navigator 2.4.2. The specific versions of the software are as follows:

- Spyder version: 5.4.3 (conda)
- Python version: 3.11.4 64-bit
- Qt version: 5.15.2
- PyQt5 version: 5.15.7
- Operating System: Darwin 22.5.0

Whilst the algorithms were being run, all cores were exploited, and CPU load was dedicated specifically towards enhancing the performance of the algorithms.

3. Experimental Components

3.1 Null Hypothesis

The algorithms, Monte Carlo Tree Search and Minimax will take different number of runs (or trials/time) to beat the user and their running time between consecutive iterations will be different as well.

3.2 Measuring Time and Complexity

Time was measured using the number of iterations or playouts that the algorithms go through. A single iteration would represent a complete cycle from selection, expansion, simulation, and backpropagation. Randomly generated data was used through each iteration.

Number of Trials Performed: 25

Big-Oh (Complexity)

- MCTS: $O\ (c.n.m)$
    - c is the constant representing the computational cost per iteration
    - n is the number of iterations performed
    - m is the computational cost per simulation
- Minimax: $O\ (b^d)$
    - b is the average branching factor, representing the average number of moves available to a player at each turn

- *d* is the depth of the game tree that the algorithm needs to search to evaluate the best move

Independent Variable: Player's Moves - in the context of analysing or implementing game strategies, the moves/decisions made by the player (either the user or AI) during the game.

Dependent Variable: Game Outcome – the result or the outcome of the game based on the decisions/moves made by the players. For this game, could include a win, loss, or a draw.

3.3 Monte Carlo Tree Search (Code & Outcome)

Figure 2: Section of the MCTS code in python depicting the main methods used.

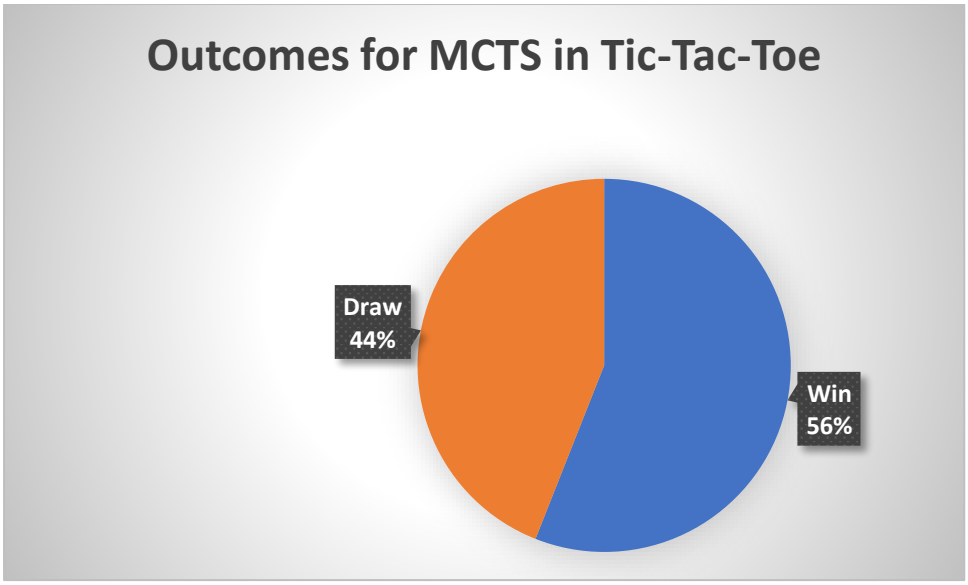

Figure 3: Pie chart depicting the outcomes after 25 trials for the MCTS algorithm in tic-tac-toe.

3.4 Minimax (Code & Outcome)

/Users/shaivan/Project 2 SCICOMP302/minimax.py

minimax.py | monte_carlo_tree_search.py | tictactoe.py

```
105
106     def set_move(x, y, player):
107         """
108         Set the move on board, if the coordinates are valid
109         :param x: X coordinate
110         :param y: Y coordinate
111         :param player: the current player
112         """
113         if valid_move(x, y):
114             board[x][y] = player
115             return True
116         else:
117             return False
118
119
120     def minimax(state, depth, player):
121         """
122         AI function that choice the best move
123         :param state: current state of the board
124         :param depth: node index in the tree (0 <= depth <= 9),
125         but never nine in this case (see iaturn() function)
126         :param player: an human or a computer
127         :return: a list with [the best row, best col, best score]
128         """
129         if player == COMP:
130             best = [-1, -1, -infinity]
131         else:
132             best = [-1, -1, +infinity]
133
134         if depth == 0 or game_over(state):
135             score = evaluate(state)
136             return [-1, -1, score]
137
138         for cell in empty_cells(state):
139             x, y = cell[0], cell[1]
140             state[x][y] = player
141             score = minimax(state, depth - 1, -player)
142             state[x][y] = 0
143             score[0], score[1] = x, y
144
145             if player == COMP:
146                 if score[2] > best[2]:
147                     best = score  # max value
148             else:
149                 if score[2] < best[2]:
150                     best = score  # min value
151
152         return best
```

Figure 4: Section of the Minimax code depicting the main methods used.

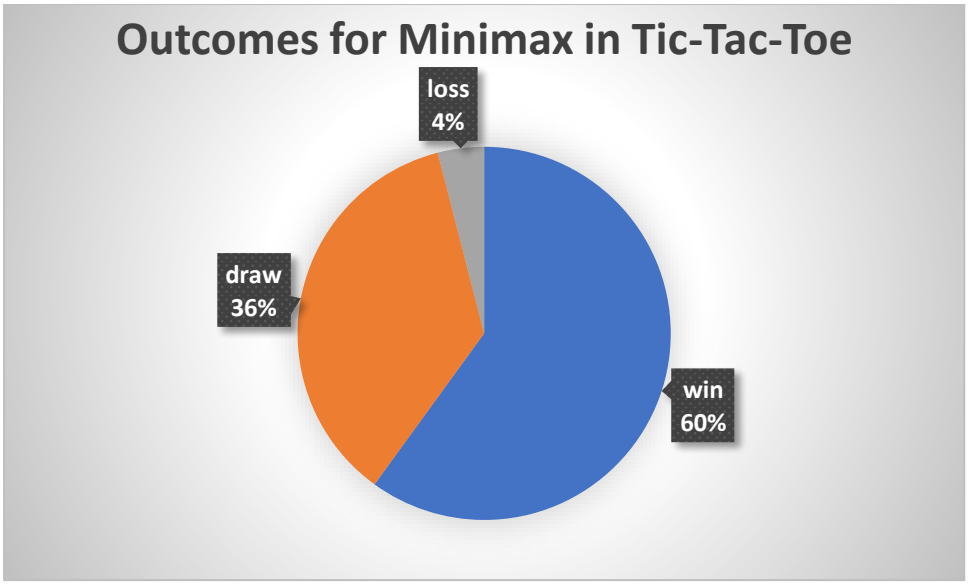

Figure 5: Pie chart representing the outcomes for minimax algorithm after 25 runs for the game tic-tac-toe.

3.5 Static Analysis & Refactoring

3.5.1 MCTS

Through Source > Run code analysis, static analysis can be performed for the python file using *pylint*. Refactored code is submitted as a separate file with the rest of the files.

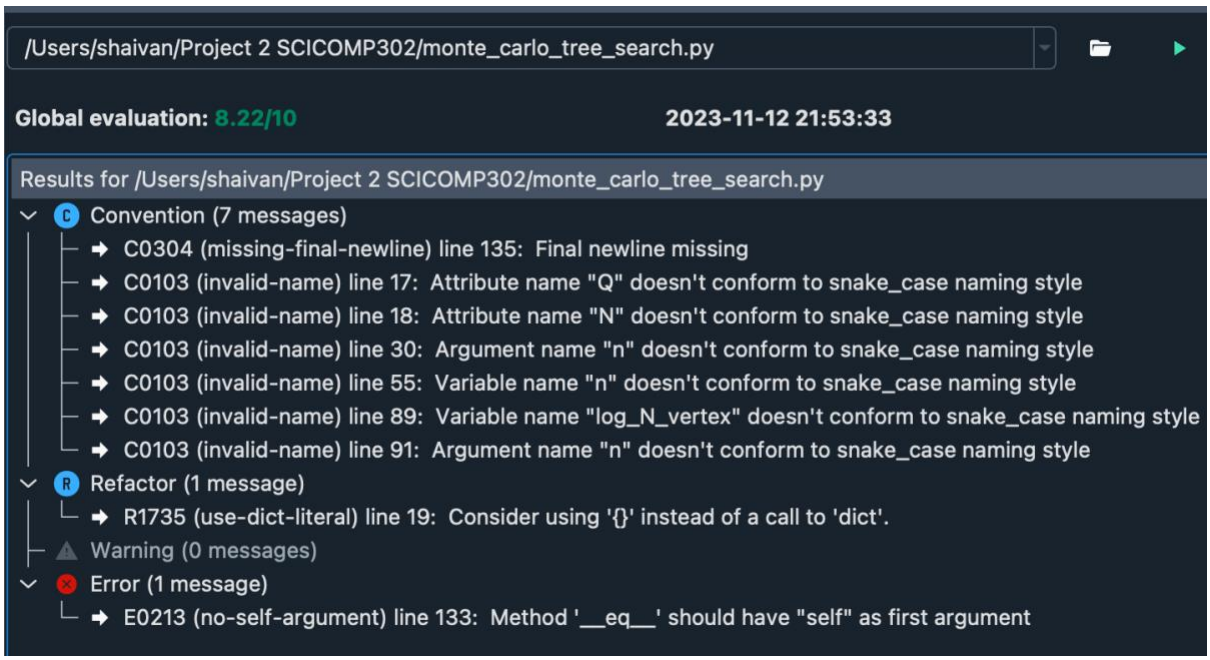

Figure 6: Resulting errors for MCTS after static code analysis through pylint.

Most of the errors are convention errors which requires to name the attributes, variables, and arguments according to the python naming convention. The refractor error can be ignored. However, the error message can lead to issues with implementation. Adding an argument “self” which has been defined before to the “eq” method will remove this error.

Refactoring would involve restructuring the code to enhance its readability, maintainability, and/or performance without modifying its behaviour. Redundant code could be made more modular. Techniques that could be applied to refactor code could possibly include simplifying conditional statements, modularization, adhering to naming conventions, reducing complexity, and improving readability. Here are the specific changes made in the refactored code (adhering to PEP 8 standards) which improve the code’s readability, modularity and conventions whilst maintaining the behaviour of MCTS:

- `self.children` Definition: Changed the initialization of `self.children` from `dict()` to `defaultdict(set)` for cleaner code. This change ensures that nodes are automatically added when needed, and the values are sets to manage node children efficiently.
- `_select` Method: In the `_select` method, adjusted the logic to append the unexplored child directly to the `path` list, enhancing code readability and reducing complexity.
- `_simulate` Method: Refactored the `_simulate` method by changing the logic to use a `while` loop to determine the reward until the node becomes terminal. This simplifies the logic and improves readability.
- `_score` Method: Created a new `_score` method to calculate the score for a node. This modification enhances readability and separates concerns by isolating the scoring logic.
- `_uct_select` Method: Improved the `uct` method within `_uct_select` by using `_score` and simplifying the calculation for the Upper Confidence Bound for Trees (UCT).
- `__eq__` Method: Corrected the `__eq__` method in the `Node` class to accept `self` and `other` as parameters, ensuring proper comparison between nodes.

3.5.2 Minimax

After running the static code analysis through the source tab with *pylint*, these are the resulting errors we find. Most are convention errors which can be modified in a simple manner to avoid them.

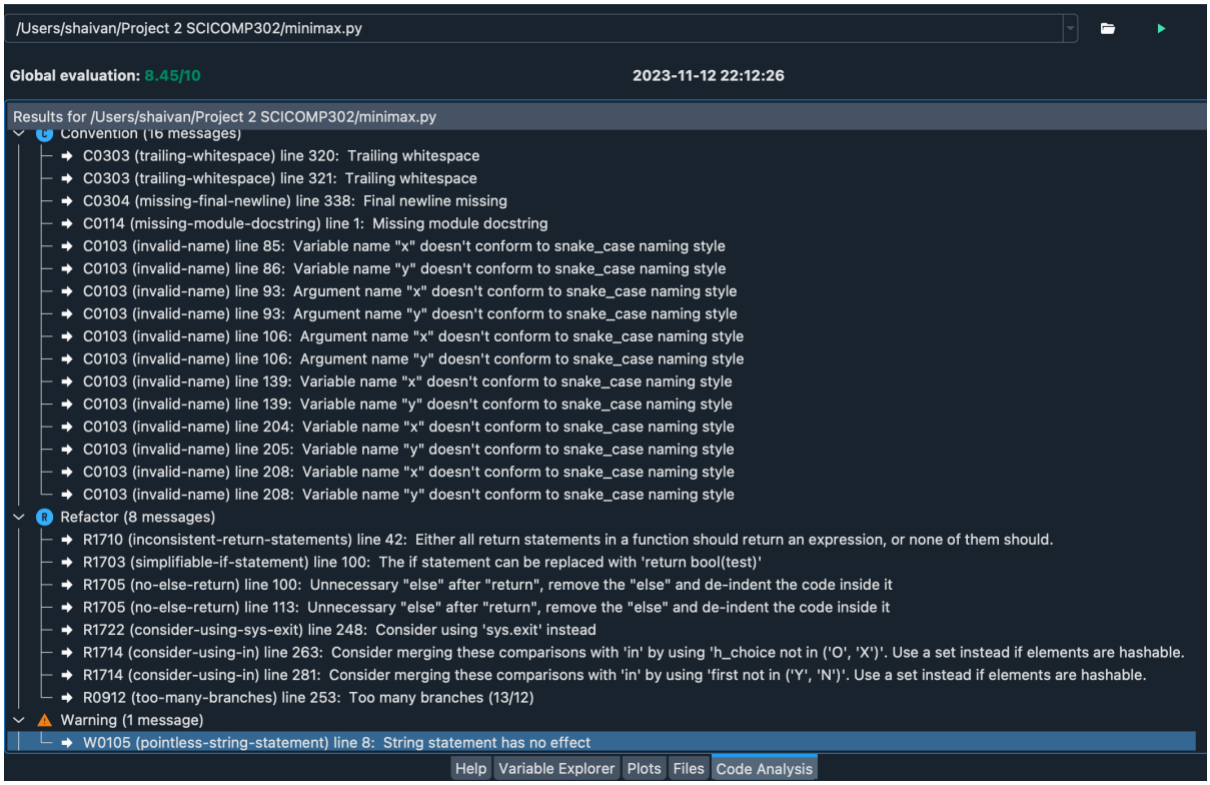

Figure 7: Resulting errors after conducting static code analysis for Minimax using pylint.

Changes made in the code to allow refactoring include:

1. Imports and Constants:

- Imported necessary modules directly (`os`, `time`).
- Defined `HUMAN` and `COMP` constants at the beginning.

2.`clean` Function:

- Consolidated the platform check and the clearing command into one variable for better readability.
- Used `os.system(clear_command)` instead of `exit()` for clearing the console.

3. `render` Function:

- Altered the loop structure to print the board.
- Employed a dictionary, `chars`, to map cell values to symbols for display.

4. AI and Human Turns:

- Separated the AI and human turn functions (`ai_turn` and `human_turn`) to improve readability and isolate functionalities.
- Utilized dictionaries (`moves`) to map numpad inputs to board coordinates in the `human_turn` function.

5. `main` Function:

- The primary game logic and orchestration remained unchanged.

3.6 *Conclusions*

We can accept the null hypothesis in the manner that MCTS and Minimax do indeed have a different performance rate with Minimax performing better for the same number of runs/trials. However, it is to be noted that the algorithm does include a loss as well. Many more runs of the performance and time complexity must be conducted to see if there is one algorithm better than the other in the long term as well as for more complex problems.

Refactoring of the Minimax code improved its time complexity although not by a significant portion since we only run 25 trials for a rather simple decision game. Upper Confidence Bound (UCT) should also be relied upon to balance exploration and exploitation in decision-making within the search tree. It adds a critical component used to guide the selection of nodes (possible decisions) within the tree. MCTS offers domain agnostics, ability to halt it at any time, and asymmetric tree growth. However, it requires a huge memory load after a few iterations, while not being the most reliable algorithm in a few problems. Minimax on the other hand offers optimal decision making, a complete search and is more applicable. However, it is expensive in larger trees with memory and complexity demands in more complex games.

References:

https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/

https://medium.com/0xcode/the-monte-carlo-tree-search-mcts-algorithm-and-machine-intuition-in-games-a9df33a6e30e

https://philippmuens.com/minimax-and-mcts

https://cs.stackexchange.com/questions/51726/whats-the-time-complexity-of-monte-carlo-tree-search

https://blogs.cornell.edu/info2040/2022/09/13/56590/

https://builtin.com/machine-learning/monte-carlo-tree-search

https://www.baeldung.com/java-monte-carlo-tree-search

https://github.com/Cledersonbc/tic-tac-toe-minimax/blob/master/py_version/minimax.py

https://gist.github.com/qpwo/c538c6f73727e254fdc7fab81024f6e1

https://www.pylint.org/