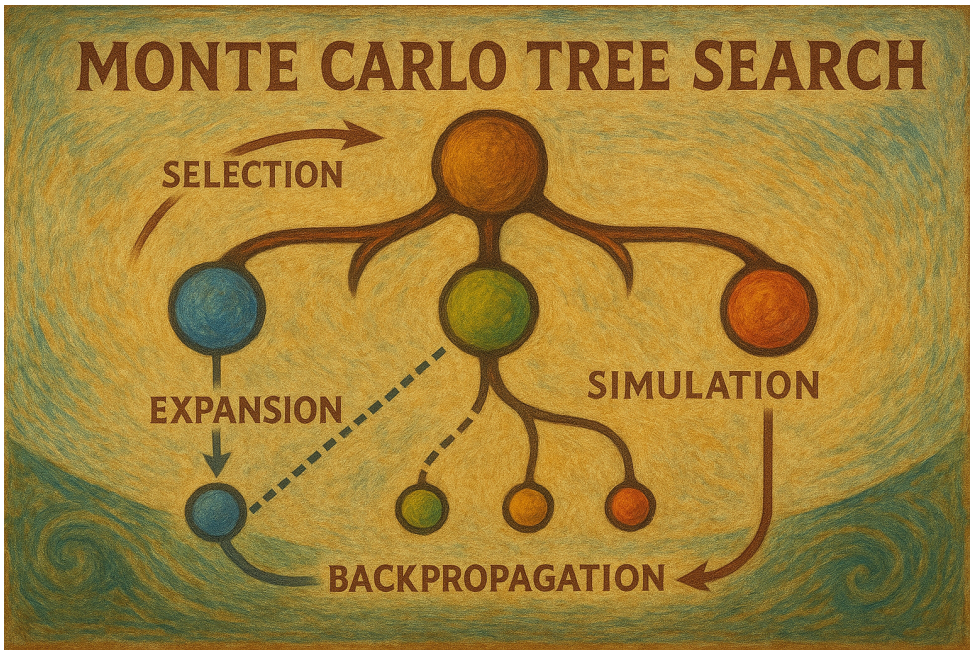
INFO-6205 PROJECT REPORT

GITHUB REPOSITORY

MONTE CARLO TREE SEARCH

Monte Carlo Tree Search (MCTS) is a clever decision-making algorithm that explores possible future moves by repeatedly simulating random outcomes. It starts from the current state, tries out different paths by playing random moves to see what happens, and then gradually focuses on the most promising ones. Over time, it builds a tree of options, using what it learns from these simulations to make smarter, more informed choices.



STEP1-SELECTION: MCTS basically begins from the current situation or state of the game, let us call this the root, it then picks moves to follow based on what's previously looked good or previously been favorable, but it also makes sure to explore moves it hasn't tried as much yet. The basis for this calculation is usually done with a formula called UCT (Upper Confidence bounds applied to Trees).

STEP2-EXPANSION: When the MCTS algorithm in its iteration hits a point in the tree where it hasn't tested all possibilities, it adds one or more new options to explore further and by this it expands the tree by adding new nodes.

STEP3-SIMULATION: The next step involves the algorithm quickly running through a random scenario, let us call this a play-out, from the new move it just added, going all the way until it reaches one of the following outcomes- a win, a loss, or some endpoint in the game.

STEP4-BACK PROPAGATION: The final step of the algorithm involves taking whatever it learned from that random scenario and updating all the moves it went through and making sure that the future decisions take this new information into account or are made taking this future information into account.

With multiple iterations the MCTS gradually figures out which paths in the tree lead to better outcomes and in context of the game the most beneficial next move. Over time, it naturally focuses on the most promising moves, helping it make effective choices even when the number of possible moves is huge.

TIC-TAC-TOE:

Tic Tac Toe is a two-player game played on a 3×3 grid. Players take turns marking X or O in empty squares. The first to get three in a row (horizontally, vertically, or diagonally) wins. If all squares are filled without a winner, the game is a draw.

OTHELLO: Othello is a two-player strategy board game played on an 8×8 grid. Players take turns placing discs that are black on one side and white on the other. The goal is to trap your opponent's discs between two of your own, flipping them to your color. The player with the most discs of their color at the end wins.

CLICK FOR RULES

MCTS IMPLEMENTATION FOR TIC-TAC-TOE

Our implementation to simulate a game of tic-tac-toe combines Monte Carlo Tree Search (MCTS) with domain-specific heuristics, what we mean by domain-specific heuristics are specialized rules and evaluation techniques that incorporate expert knowledge about a particular problem and in this instance it is the game of tic-tac-toe. Our approach consists of two key components:

DIRECT THREAT DETECTION: In this step before running computationally expensive search, we have implemented the algorithm in such a way that it first checks for immediate winning moves or blocking opportunities by examining all rows, columns, and diagonals for two-in-a-row patterns with an empty third position. This step ensures that optimal play in critical situations without relying on statistical sampling is carried out.

ENHANCES MCTS PROCESS: For moves that are less obvious in nature, the algorithm carries out for multiple iterations a four-phase MCTS approach:

- ➔ **SELECTION**: Traverses the game tree using UCB (Upper Confidence Bound) formula to balance exploration and exploitation
- ➔ **EXPANSION**: Creates nodes for all valid moves from the selected position
- ➔ **INTELLIGENT SIMULATION**: Rather than purely random playouts, the simulation prioritizes winning and blocking moves before falling back to random play
- ➔ **BACK PROPAGATION**: Updates win statistics throughout the tree, alternating perspective at each level

This hybrid approach that we have implemented to simulate the game of tic-tac-toe combines the strengths of traditional game-specific heuristics and also leverages the statistical strength of MCTS, resulting in an ideal play with minimal computational resources. Our implementation demonstrates how the use of domain knowledge can significantly enhance the performance of search algorithms, a principle that scales and can be applied to more complex games.

MCTS IMPLEMENTATION FOR OTHELLO

Using similar ideas as we did for tic-tac-toe we implemented an algorithm that combines Monte Carlo Tree Search (MCTS) with Othello-specific heuristics to create an effective AI player of the game. Just like we saw in our approach to tic-tac-toe, our implementation for the game of othello also has two key components:

STRATEGIC PATTERN RECOGNITION: Before running the full search, our algorithm carries out a check for high-value corner moves, as these are known to be strategically dominant in the game of Othello. This optimization allows for immediate recognition of strong positions without requiring extensive sampling.

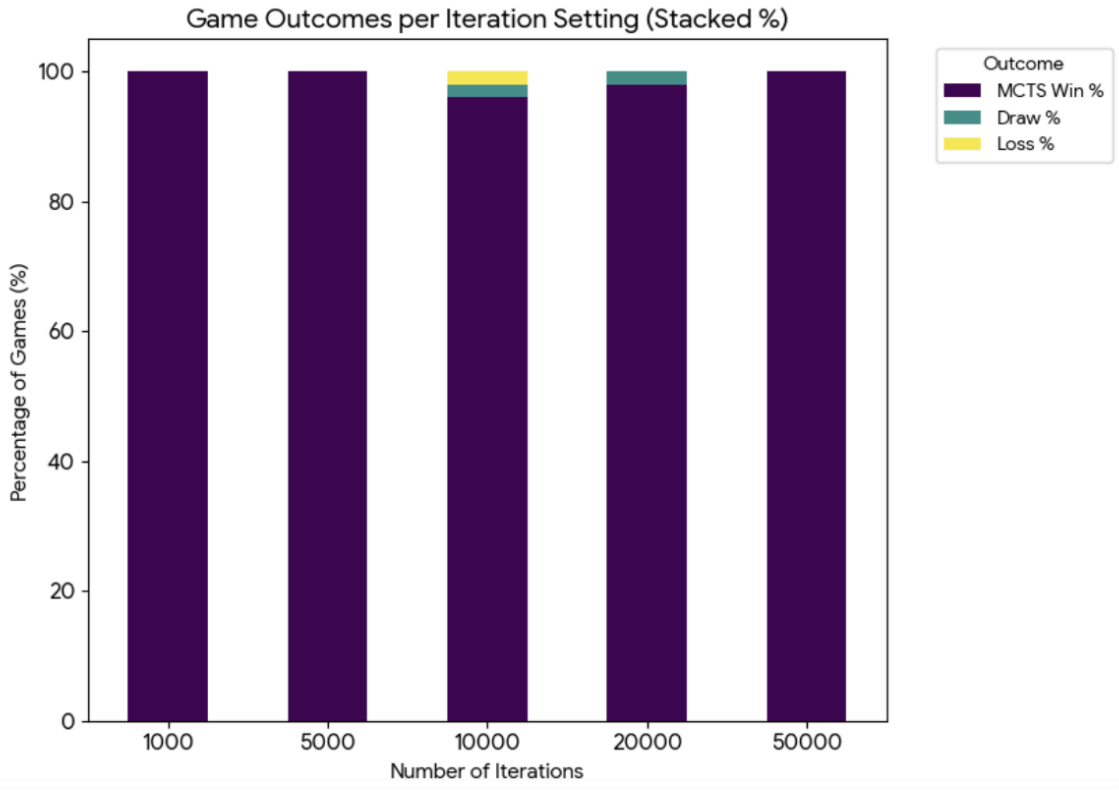
ENHANCED MCTS PROCESS: For standard positions, the algorithm employs almost the same four phase MCTS approach with an improved simulation

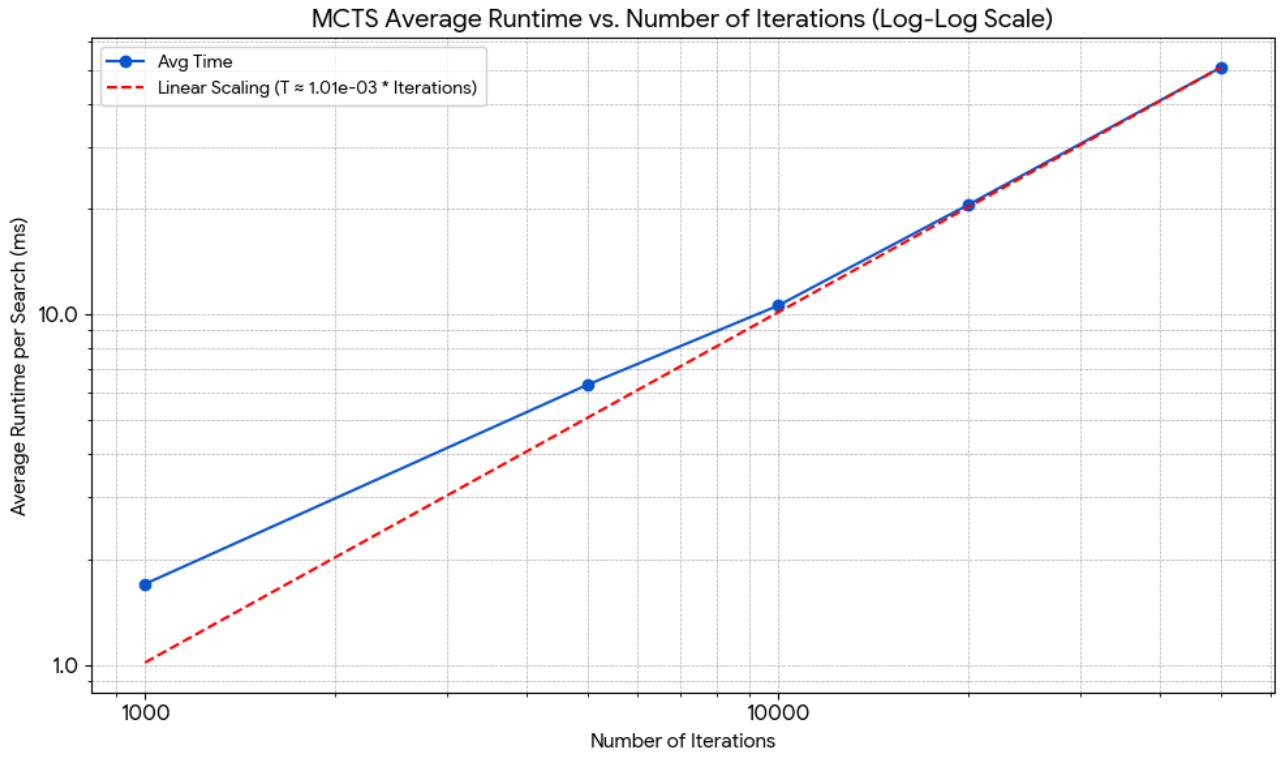
- ➔ **HEURISTIC-GUIDED SIMULATION**: Uses a 90/10 blend of strategic and random play during simulation, evaluating moves based on position weights, piece count, and opponent mobility.Our implementation incorporates the domain knowledge by making use of a position weights matrix that values corners quite heavily (100 points) and penalizes dangerous "X-squares" which are adjacent to unoccupied corners (-50 points). This knowledge-enhanced approach allows the algorithm to play strategically sound moves with relatively few iterations, balancing computational efficiency with strong gameplay.

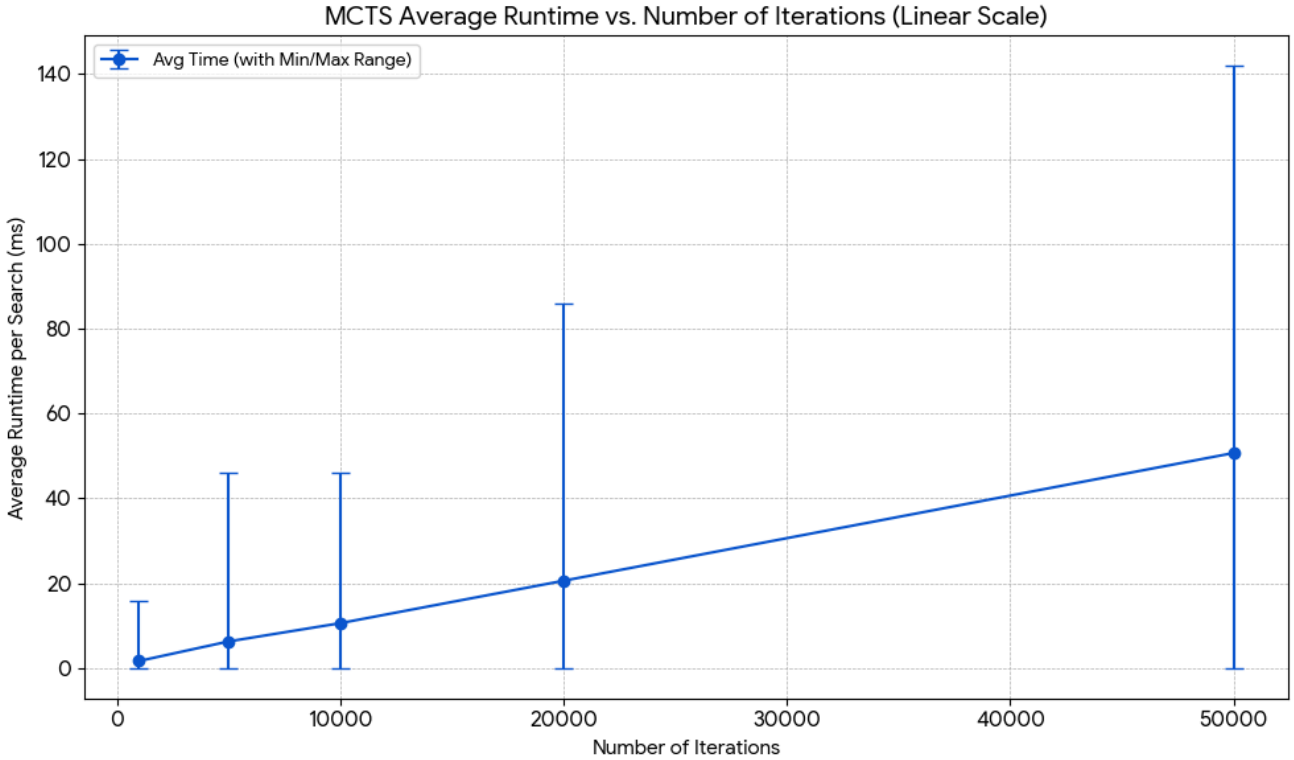
MCTS BENCHMARKING AND STATISTICS FOR TIC-TAC-TOE:

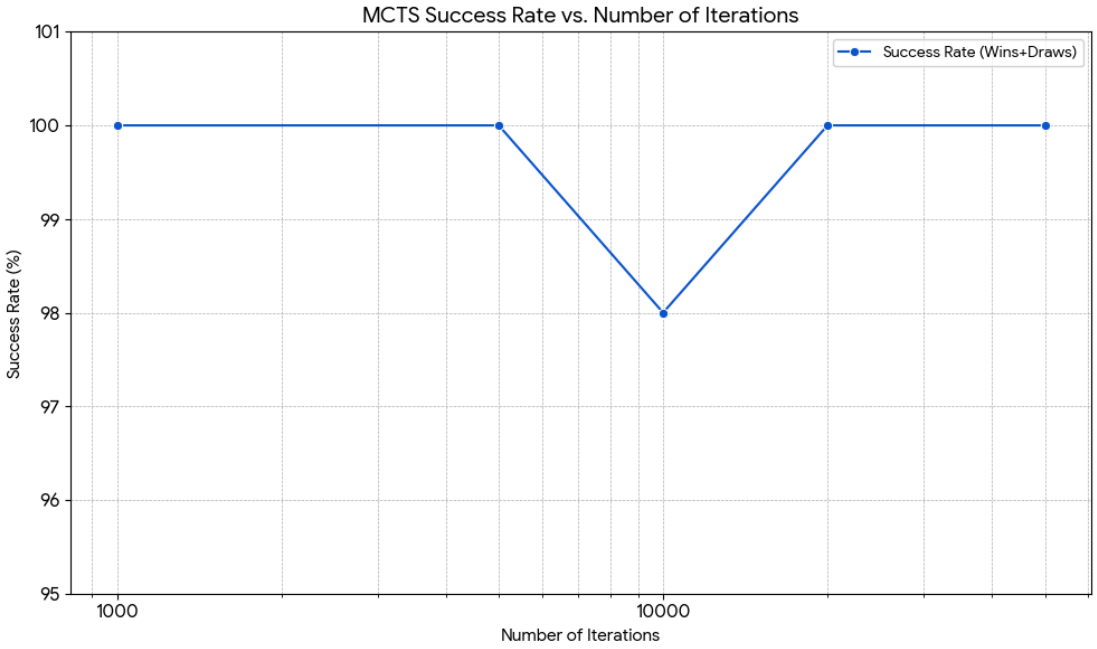
When we benchmarked implementation of the MCTS algorithm for the simulation of tic tac toe we observed that it demonstrates a very high effectiveness in this benchmark, we observed the algorithm achieving near-perfect performance across all tested iteration counts, ranging from 1,000 to 50,000 iterations per search. We noticed that increasing iterations did slightly improve consistency by avoiding the single loss and draw observed at 10,000 iterations, the gains in win/success rate beyond the initial levels were marginal. We also observed that the computational cost, measured by average runtime per search, scaled almost perfectly linearly with the number of iterations and this can be confirmed by our log-log plot. We can see that by doubling the iterations we observed roughly doubled the average search time. Through our observations, the primary conclusion is that while MCTS performs exceptionally well here, there's a clear trade-off: increasing iterations yields substantial increases in runtime for potentially negligible improvements in win rate in this context.

	Iterations	MCTS Wins	Draws	Losses	Success Rate (%)	Avg Time (ms)
0	1000	50	0	0	100.0	1.7
1	5000	50	0	0	100.0	6.3
2	10000	48	1	1	98.0	10.6
3	20000	49	1	0	100.0	20.6
4	50000	50	0	0	100.0	50.7




MCTS Average Runtime vs. Number of Iterations (Log-Log Scale)
Avg Time
Linear Scaling (T ≈ 1.01e-03 * Iterations)
Average Runtime per Search (ms)
10.0
1.0
1000
10000
Number of Iterations


MCTS Average Runtime vs. Number of Iterations (Linear Scale)
Avg Time (with Min/Max Range)
Average Runtime per Search (ms)
140
120
100
80
60
40
20
0
0
10000
20000
30000
40000
50000
Number of Iterations



MCTS BENCHMARKING AND STATISTICS FOR OTHELLO:

When we benchmarked the implementation of the MCTS algorithm for the simulation of Othello games against a random player, we observed that it demonstrates significant effectiveness, clearly learning to defeat the random opponent much more often than not. Unlike benchmarks in simpler solved games, the performance was not near-perfect, with success rates (Wins + Draws) ranging from 55% to a peak of 75% in this test.

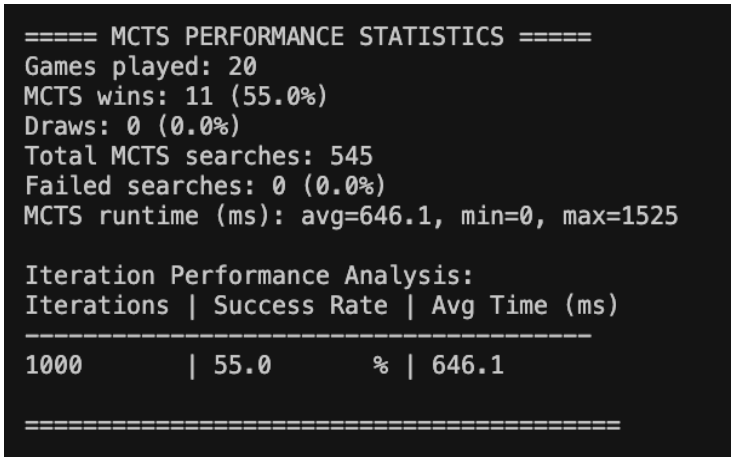
We noticed a substantial improvement in performance when increasing iterations from 1,000/5,000 (55% success) up to 10,000 (75% success), indicating that more search effort significantly improved strategic play in the early stages. However, beyond this point, further increasing iterations to 20,000 and 50,000 resulted in the success rate plateauing and slightly decreasing (70%), indicating that the gains became negligible or potentially negative at higher search depths in this specific context.

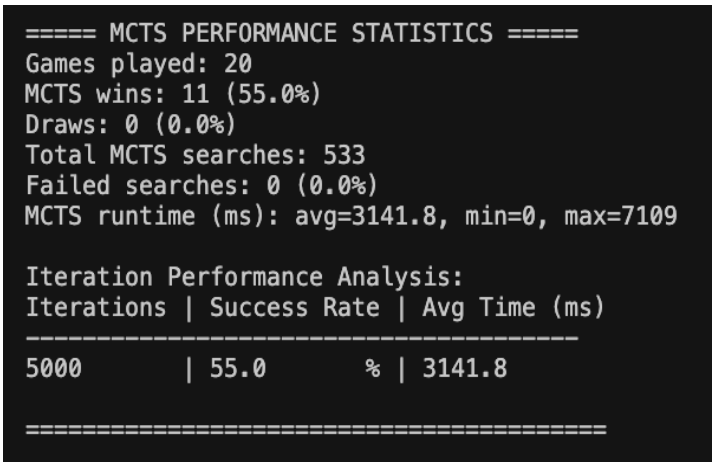
During our experimentation we also observed that the computational cost of carrying out MCTS for different iterations, measured by average runtime per search, scaled almost perfectly linearly with the number of iterations. This was confirmed by our log-log plot, which showed that doubling the iterations roughly doubled the average search time, aligning with theoretical expectations.

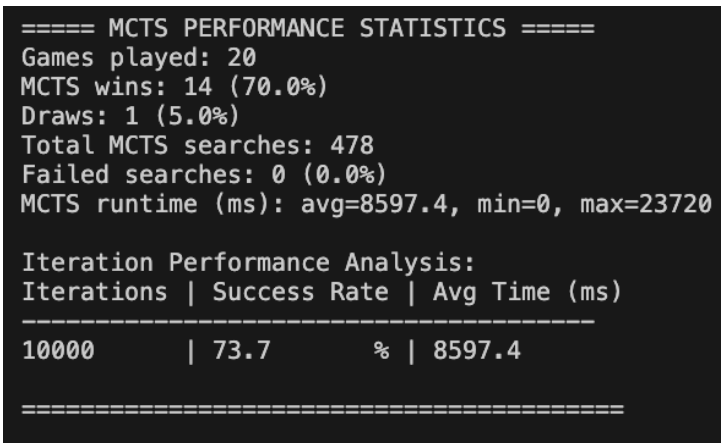
Through our experimentation and the results obtained we can make the following conclusions for the othello MCTS algorithm against a random opponent: while MCTS is effective and benefits from increased computation initially, there's a crucial trade-off: increasing iterations up to ~10,000 provides significant performance improvements at a clear runtime cost, but further increases yield negligible (or slightly negative) performance returns while substantially increasing runtime further. This suggests an optimal

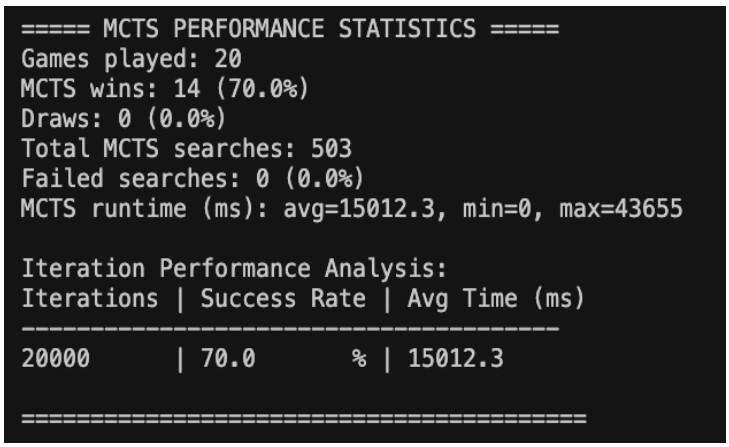
iteration count for our implementation is around 10k for balancing performance and efficiency in this specific benchmark scenario.

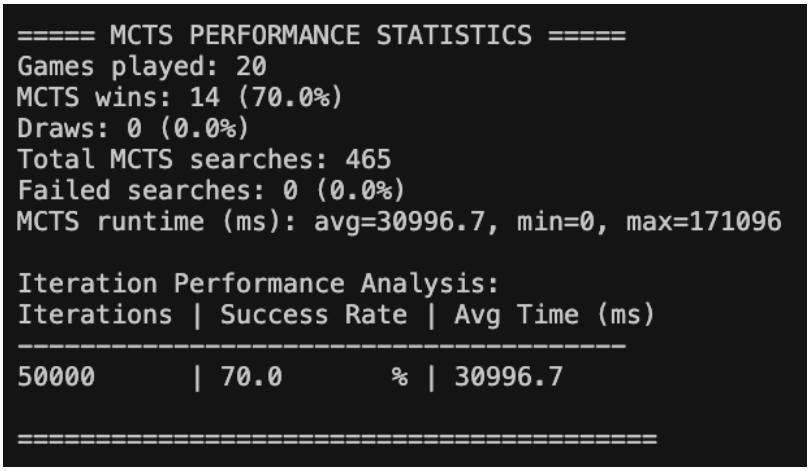
	Iterations	MCTS Wins	Draws	Losses	Success Rate (%)	Avg Time (ms)
0	1000	11	0	9	55.0	646.1
1	5000	11	0	9	55.0	3141.8
2	10000	14	1	5	75.0	8597.4
3	20000	14	0	6	70.0	15012.3
4	50000	14	0	6	70.0	30996.7

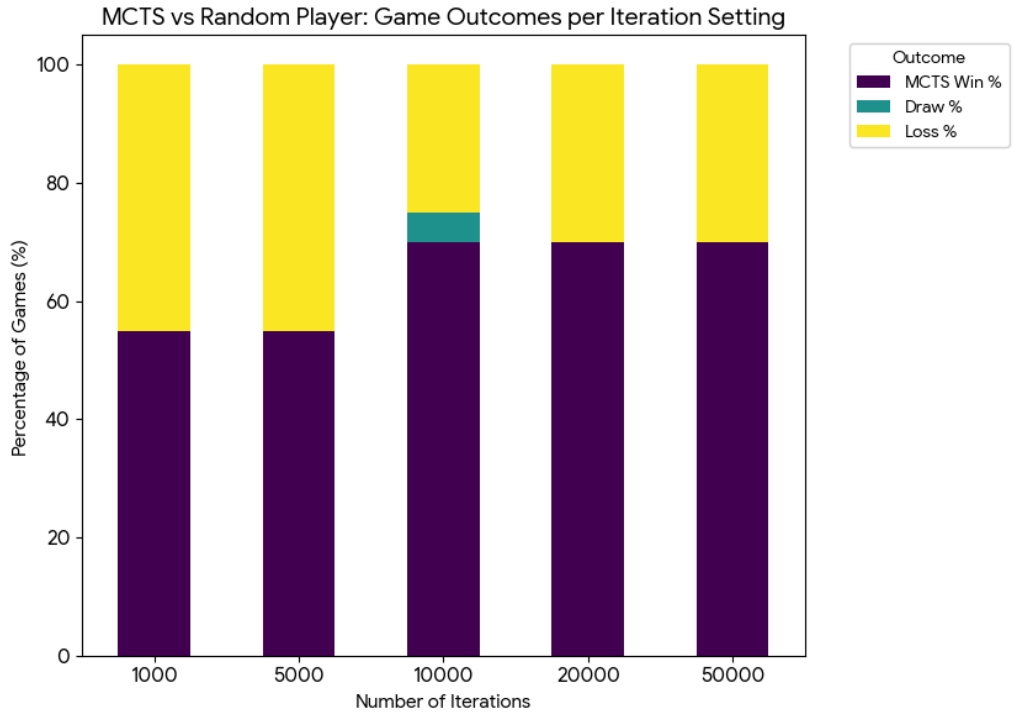


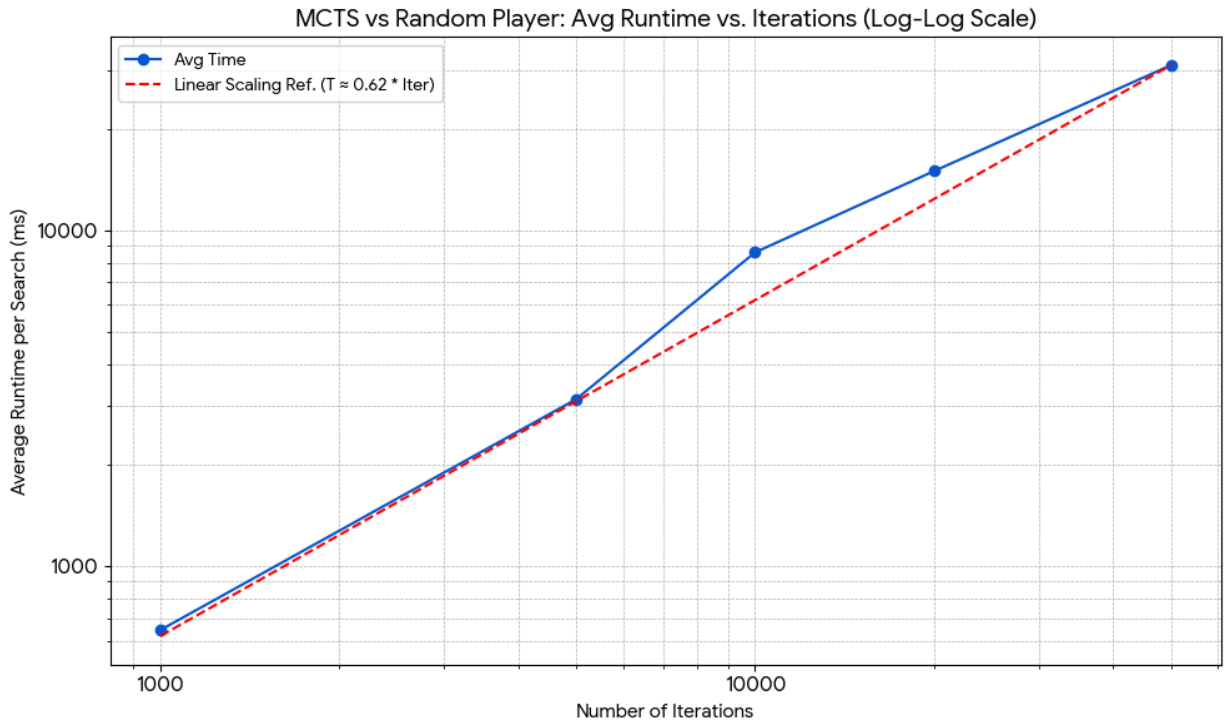


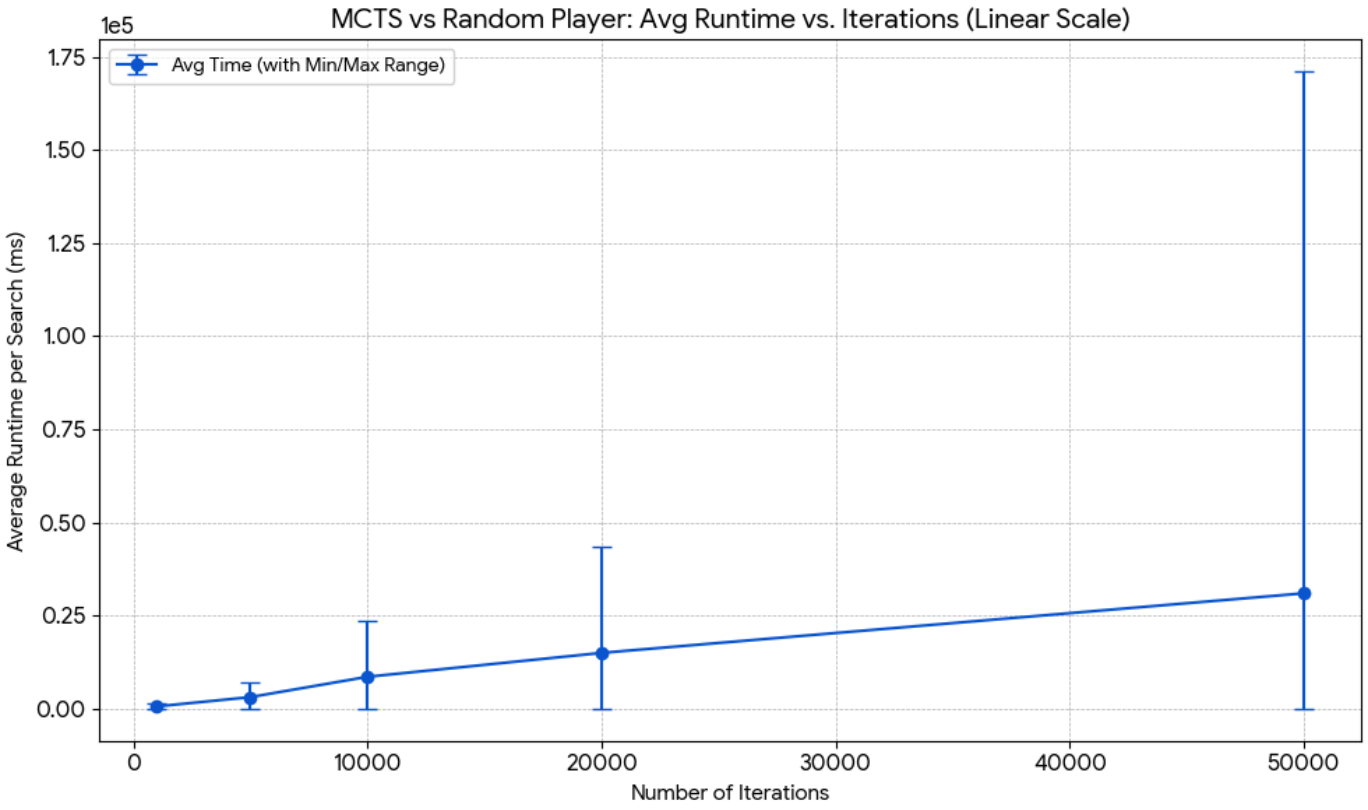


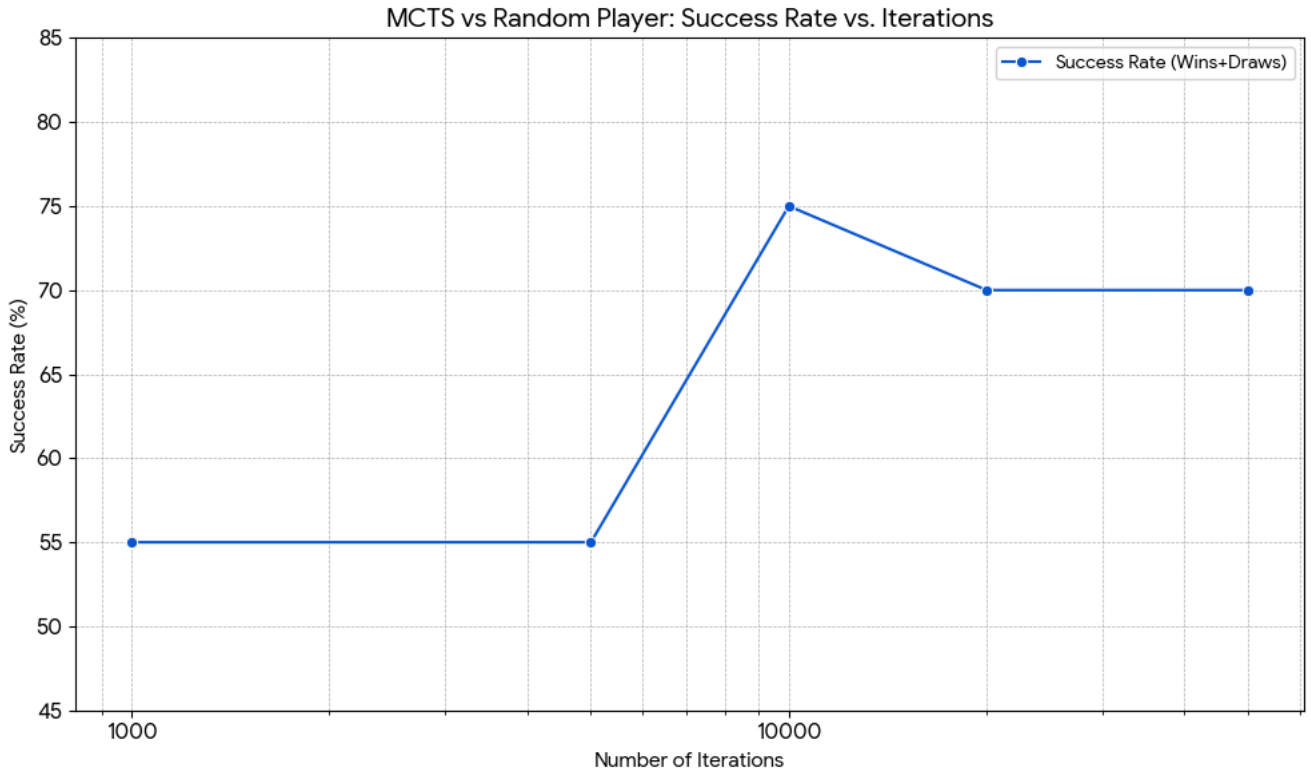





MCTS vs Random Player: Game Outcomes per Iteration Setting
Outcome
MCTS Win %
Draw %
Loss %
Percentage of Games (%)
0
20
40
60
80
100
1000
5000
10000
20000
50000
Number of Iterations


MCTS vs Random Player: Avg Runtime vs. Iterations (Log-Log Scale)
Avg Time
Linear Scaling Ref. (T ≈ 0.62 * Iter)
Average Runtime per Search (ms)
10000
1000
1000
10000
Number of Iterations


MCTS vs Random Player: Avg Runtime vs. Iterations (Linear Scale)
1e5
Avg Time (with Min/Max Range)
Average Runtime per Search (ms)
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0
10000
20000
30000
40000
50000
Number of Iterations


MCTS vs Random Player: Success Rate vs. Iterations
Success Rate (Wins+Draws)
Success Rate (%)
85
80
75
70
65
60
55
50
45
1000
10000
Number of Iterations

TEST COVERAGE