1.5em 0pt

GIST AGI Spring 2025 - HW 1

Making an Othello AI agent using a simplified version of the MCTS algorithm

20258009, Matouš Bohoněk, bohonmat@gm.gist.ac.kr

The goal of this project was to create a working AI agent for the game `Othello` in JavaScript. The implementation of the algorithms discussed can be found in the `source.js` file.

> **Problem 1: Using a Simplified Version of MCTS**
>
> Initially, my goal was to implement a classic Monte Carlo Tree Search (MCTS) with Upper Confidence Bound (UCB) and Rapid Action Value Estimation (RAVE). However, due to the updated time limit of approximately 50 ms per decision and poorer performance, I resorted to a simplified version of MCTS.

1 General Approach

The standard MCTS iteratively builds a game tree, balances exploration and exploitation, and starts random playouts from known leaf nodes. My simplified strategy starts random playouts from every possible next move, equally distributing time between all of them. See the following pseudocode:

Algorithm 1 Monte Carlo Move Selection

Require: *board*, *player*, *opponent*, *timeLimit*
timePerMove ← *timeLimit*/number of *validMoves*
bestMove ← null
bestWinRate ← null
for each *move* in *validMoves* **do**
    startTime ← CurrentTime()
    MakeMove(*move*)
    wins ← 0
    games ← 0
    while CurrentTime() - *startTime* < *timePerMove* **do**
        if RandomPlayout() = WIN **then**
            wins ← *wins* + 1
        end if
        games ← *games* + 1
    end while
    RevertMove(*move*)
    winRate ← *wins*/*games*
    if *winRate* > *bestWinRate* **then**
        bestMove ← *move*
        bestWinRate ← *winRate*
    end if
end for
return *bestMove*

As you can see, most of the time is spent repeatedly calling the `RandomPlayout()` function. This function randomly selects valid moves until the game is over and returns the result.

1.1 Two-Phase Approach

One slight improvement that I made to the general idea is to filter out bad moves when we have too many options. When the agent has five or more available moves, we first run MCTS for 40% of the given time and take only the top three moves. Then, we run the MCTS with the remaining 60% of the time for only these three moves. This improves the exploration of more promising moves by first discarding the worst moves.

2 Heuristics Implemented

Implementing heuristics in a Monte Carlo algorithm is not as straightforward as in other algorithms such as Minimax. The reason is that, while Minimax always benefits from a better state evaluation, MCTS relies on statistical sampling. Unless we have a perfect heuristic, deterring the algorithm from sufficiently sampling all possible moves might prevent it from discovering better solutions.

The main goal of any Monte Carlo algorithm is to sample as much as possible from the given environment. On the other hand, having at least some quick guiding heuristic has proven to be beneficial from my testing. The requirement for the heuristic is that it is easy to compute because we need to execute as many moves as possible.

2.1 Positional Heuristic

The only heuristic that has proven beneficial is assigning weights to different positions on the board. The idea behind the weight matrix is simple: incentivize taking corners, then edges, while deterring moves neighboring the edges. Here is what the matrix looked like:

$$\begin{bmatrix} 100 & 10 & 80 & 40 & 40 & 80 & 10 & 100 \\ 10 & 8 & 20 & 20 & 20 & 20 & 8 & 10 \\ 80 & 20 & 40 & 40 & 40 & 40 & 20 & 80 \\ 40 & 20 & 40 & 40 & 40 & 40 & 20 & 40 \\ 40 & 20 & 40 & 40 & 40 & 40 & 20 & 40 \\ 80 & 20 & 40 & 40 & 40 & 40 & 20 & 80 \\ 10 & 8 & 20 & 20 & 20 & 20 & 8 & 10 \\ 100 & 10 & 80 & 40 & 40 & 80 & 10 & 100 \end{bmatrix}$$

Usually, in the literature, you can find negative values for positions neighboring the edges, but because of my implementation of weighted random move selection, I had to use non-negative values.

3 Performance Against Built-In Strategies

The main benefit of this algorithm is its time versatility. If we have the option to spend more time "thinking," we can leave the algorithm running longer to generate better moves. However, due to its non-deterministic nature, it is not guaranteed that doubling the available time will always yield better gameplay. From my testing, even the 50 ms time limit was enough to reliably beat all

built-in strategies every time. I also ran tournaments with different time limits, ranging from 20 ms to 1 s. As expected, the 20 ms strategy performed the worst. Surprisingly, the 1 s strategy was beaten by the 0.5 s strategy, and the 50 ms strategy outperformed both 0.1 s and 0.2 s. This could be a statistical anomaly, and with more games played, performance might correlate more closely with time limits. However, this shows that doubling the time limit does not necessarily double the performance.

4 Challenges and Future Improvements

4.1 Challenges and Weaknesses

One of the biggest weaknesses of the MCTS algorithm is its vulnerability to so-called traps. In some cases, a move may appear statistically promising, perhaps showing a 99% win rate in random playouts. However, this high success rate might be misleading because random simulations often encounter suboptimal responses from the opponent. A skilled player could exploit the critical 1% of scenarios where the move results in a loss, overturning the favorable outcome that MCTS predicted.

Another weakness is that during the early game, the algorithm explores far fewer games than in the endgame. This is because simulating playouts from the very beginning takes many more steps than simulating them from a state with only two moves remaining. The number of moves grows about three times in the first eight moves of the game.

4.2 Possible Improvements

The most straightforward improvement would be the addition of multithreading. This algorithm can benefit greatly from parallelization, as it does not need to deal with shared memory resources, and all threads can run independently. Also, as we observed above, while doubling the compute power doesn't necessarily mean a big performance upgrade, having eight times the compute power will likely improve performance noticeably.

Another improvement could involve adding more easy-to-compute heuristics. The reason I haven't used other heuristics is that those I tried (such as mobility, parity, and stability) took too long to compute, and their benefit was outweighed by the major reduction in the number of simulations.

Finally, incorporating a Minimax algorithm and a book of starting moves could be advantageous. As mentioned above, the number of simulations isn't very high in the first few moves of the game. Introducing a standard opening book could prevent the agent from making poor decisions early on. Later, with fewer than six or eight moves remaining (from my testing, this depth is still computable in a reasonable time), switching from MCTS to a standard Minimax algorithm could provide a perfect game finish, avoiding MCTS's vulnerability to traps at critical game stages.