Monte Carlo for Computational Game Theory - Othello

Ana Isabel Fernandez Sirgo, Sheila Whitman

April 26, 2024

1 Introduction

Othello, a timeless board game of strategy and skill, has long served as a benchmark for evaluating artificial intelligence (AI) techniques in the domain of game playing. Beneath its simple rules lies a complex web of strategic depth, making it an ideal testbed for AI systems seeking to demonstrate mastery in complex decision-making environments. In this paper, we present an exploration of Monte Carlo Tree Search (MCTS) as applied to developing an AI bot capable of proficient Othello gameplay.

Othello, also known as Reversi, challenges players to outmaneuver their opponent and dominate the board by strategically placing and flipping discs of their color. The game's dynamic nature, coupled with its large decision space, poses a formidable challenge for traditional search algorithms. Monte Carlo Tree Search (MCTS), with its ability to balance exploration and exploitation, offers a promising avenue for navigating the complexities of Othello gameplay.

At its core, MCTS embodies a principled approach to decision-making, iteratively exploring the game tree through a combination of random simulation and statistical analysis. By simulating thousands of possible game trajectories and selecting promising moves based on their expected outcomes, MCTS aims to craft winning strategies while adapting dynamically to evolving game states. Its ability to adapt and make strategic moves sets it apart, making MCTS an attractive candidate for AI-driven Othello gameplay.

2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) has emerged as a leading approach for decision-making in domains such as game playing, resource allocation, and optimization. Originally proposed for computer Go, MCTS has since been applied to various other domains, including robotics, planning, and real-time strategy games [1]. The algorithm's effectiveness lies in its ability to balance exploration and exploitation, allowing it to navigate complex state spaces efficiently.

The MCTS algorithm consists of four key steps: Selection, Expansion, Simulation, and Back-propagation. The algorithm builds a search tree at each iteration by repeatedly selecting nodes, expanding them, simulating play-outs, and updating statistics. The search tree guides decision-making by providing insights into the likely outcomes of different actions.

1. **Selection:**

   At each step, the algorithm selects a node based on a selection function that balances exploitation and exploration. Let i be the current node being considered for selection. The selection function, denoted as $UCB(i)$, is given by:

$$UCB(i) = V_i + 2C\sqrt{\frac{2lnN}{n_i}} \tag{1}$$

   where N is the number of parent visits, n_i is the number of visits to this node. The constant C is a constant that controls the balance between exploration and exploitation, where V_i is the exploitation

term, and $\sqrt{\frac{lnN}{n_i}}$ is the exploration term. The exploitation term is the average value of that state, we will provide an explanation on how to perform the calculation in the back-propagation step.

2. **Expansion:**

   Once a leaf node is reached, if the node has not been visited the algorithm does not expand it; it moves immediately to the Simulation phase. On the other hand, if the node has been visited, the algorithm expands it by adding all possible child nodes, which represent all possible moves or states. From there the algorithm selects a node to simulate.

3. **Simulation (Rollout):**

   The algorithm performs a random simulated playout from the newly added node until it reaches a terminal state. This is done a number for a limited time duration or a predetermined number of iterations. After this, the algorithm decides which is the best action.

4. **Back-propagation:**

   The algorithm updates the statistics of all nodes visited during the selection phase and their ancestors based on the simulation outcome. At each node we keep a score of visits n and game results t. When we back-propagate, at every parent node, we update these n and t values. For equation (1), the average value of the node i, V_i is calculated using these scores as follows:

$$V_i = \frac{t}{n} \tag{2}$$

Note that the algorithm will favor nodes that have never been visited before. Once those nodes have been visited, the algorithm will choose to explore the nodes that have won the game more.

In their paper "A Survey of Monte Carlo Tree Search Methods," Browne et al. discuss how the constant C in the exploration term of Monte Carlo Tree Search (MCTS) can be adjusted to control the level of exploration conducted during the search process [1]. They cite the work of Kocsis and Szepesvári, who demonstrated that setting $C = \frac{1}{\sqrt{2}}$ satisfies the Hoeffding inequality when rewards are confined within the range of [0,1]. However, Browne et al. note that a different value of C may be necessary when rewards extend beyond this range [1]. Additionally, they suggest that certain enhancements to the MCTS algorithm may perform optimally with a different value for C. For this work, we used the $C = \frac{1}{\sqrt{2}}$.

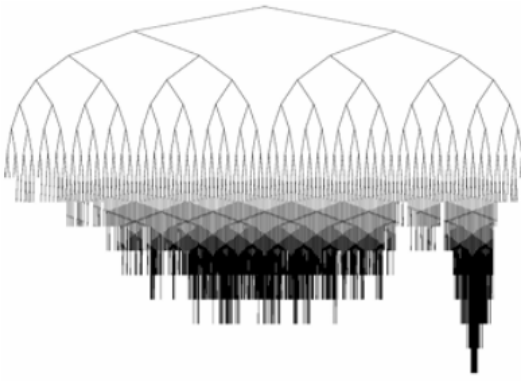

Figure 1: [1]

The process of tree selection by algorithms exhibits a bias toward nodes that are considered more promising while still maintaining some probability of selecting other nodes. As a result, the tree's structure becomes asymmetric over time, with a tendency to concentrate on regions that are perceived as more promising and significant. Browne et al. provide a diagram that illustrates this asymmetric growth of

the tree. Furthermore, the shape of the tree that emerges from this process can provide valuable insights into the nature of the analyzed game. According to Browne et al., analyzing the shape of trees can help distinguish between playable and unplayable game scenarios.

3 Tic-Tac-Toe

We first consider the simple game of Tic-Tac-Toe as a proof of concept of Monte Carlo Tree Search. Game tree complexity is one metric of analyzing the complexity of a game based on the number of leaf nodes in the smallest full-width decision tree (all nodes at each depth) that establishes the value of the initial position [2]. Tic-tac-toe has been shown to have approximately $5,478$ legal moves with a game tree complexity of $362,880$ for the simple $3x3$ board we are considering here [3].

3.1 Rules

1. Tic-tac-toe is played on a board of 3x3 squares.
2. One player plays as a cross and the other plays as a circle.
3. (In our implementation) The dark cross player plays first.
4. To play: the players alternate placing crosses and circles on the board until: (1) the board is fill or (2) there are three consecutive symbols in a row (diagonal, row, or column)

Figure 2: [3]

3.2 Coding Strategy

In this work, we are following Object-Oriented Programming to code classes for our Monte Carlo Tree Search and for our boards. This way, we only have separate classes for our two game boards and the same Monte Carlo Tree Search class to be utilized by the two games. In the previous section we explained the algorithm behind the MCTS class, so here we will focus on the functions needed to build the Tic-Tac-Toe board.

- **Initializing Class** Setting up the initial game rule, calling the initialization of the board (if game is new) or copying current board, assigning the players.
- **Initialize board** Setting up the blank board.
- **Print board** Printing the board.
- **Is Valid** Checking that the user-inputted move is not a space already taken or off of the board.
- **Make move** Draws in the circle or cross to the board in the defined position.
- **Is Win** Checks if a player won the game (3 consecutive symbols).
- **Is Tie** Checks if the two players ran out of moves with no winner.

4 Othello

4.1 History

The history of Othello dates back to 1883 when it was originally called Reversi. In 1971 the modern version of the game was patented in Japan with a set initial state of the board. The game is a two-player game where each player aims to have more of their colored disks on the board. [4] Othello has been shown to have at most 10^{28} legal moves in the full size $8x8$ board with a game tree complexity of 10^{58} [4, 5]. The $4x4$ and $6x6$ boards show a bias to the second player under perfect play [4, 6]. Just this year, in his paper, "Othello is Solved," Hiroki Takizawa describes how he "weakly solved" the $8x8$ board by utilizing alpha-beta pruning to match results with the expected number of legal moves (10^{28}) with both players resulting in a tie. In this work, we only explore building Othello with Monte Carlo Tree Search, but we note that alpha-beta pruning would be another (and more promising for perfect play) algorithm for developing an Othello playing bot.

4.2 Rules

1. Othello is played on a board of 8x8 squares.
2. There are 68 identical pieces - each a disk colored on one side white and the other side black. Each player starts with 32 pieces and one player plays as light (white side up) and the other plays as dark (black side up).
3. Initially, the board is set with the additional 4 disks in the middle, 2 white along the diagonal and two black across the diagonal.
4. The dark player plays first.
5. To play: The player going must place their piece (their side up) in such a way that they create a connected straight line, either horizontally, vertically, or diagonally across from another one of their pieces (their side up) with the opponent's pieces (opponent side up) in between. Once placed, all of the opponents pieces, which fall between the two players pieces, are flipped to be the player's color. This is repeated until the game is over.

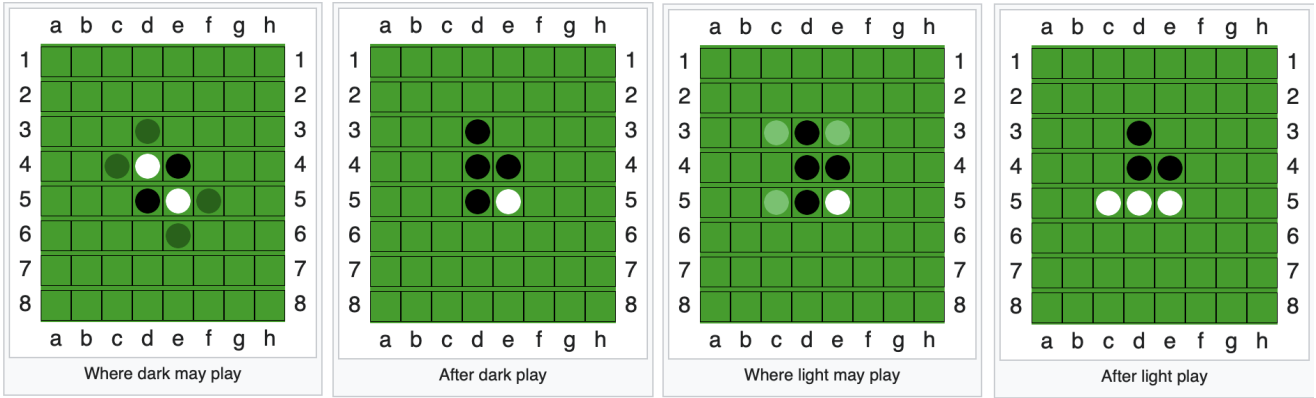

Figure 3: [4]

6. If one player cannot make a valid move, play passes back to the other player.
7. If neither player can move, the game is over, either from running out of space on the board or from running out of legal moves for both players.
8. The player with the most pieces on the board at the end of the game wins.

9. If the game ends with both players having an equal number of pieces on the board, it is a tie.

4.3 Coding the Game

As we did with Tic-tac-toe, we will focus on the functions needed to build the Othello board. While the game appears simple to play, coding the board proved to be difficult. Specifically, implementing the skipping of players effected the initial implementation of MCTS.

- **Initializing Class** Setting up the initial game rule, calling the initialization of the board (if game is new) or copying current board, assigning the players.
- **Initialize board** Setting up the blank board.
- **Print board** Printing the board.
- **Is Valid** Checking that the user-inputted move is not a space already taken, off of the board, or not a valid move for the player to make.
- **Make move** Places the white or black disk in the chosen location, then flips the other disks as applicable.
- **Is Win** Checks if a player won the game: (1) no moves left and scores are different (2) both players are out of moves and scores are different.
- **Is Tie** Checks if a players tied: (1) no moves left and scores are the same (2) both players are out of moves and scores are the same.

Additionally some helper functions were needed to help decide if moves were indeed valid, skipping turns, and obtaining the correct ways to flip the disks.

5 Code Availability

As this project's main objective was to build a MCTS bot to play against in both Tic-Tac-Toe and Othello, we publicly share our codes on Github: `https://github.com/whishei/MCMC-Project`

References

[1] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, S. Colton, A survey of monte carlo tree search methods, IEEE Transactions on Computational Intelligence and AI in games 4 (1) (2012) 1–43.

[2] Wikipedia contributors, Game complexity — Wikipedia, the free encyclopedia, `https://en.wikipedia.org/w/index.php?title=Game_complexity&oldid=1171904906`, [Online; accessed 22-April-2024] (2023).

[3] Wikipedia contributors, Tic-tac-toe — Wikipedia, the free encyclopedia, `https://en.wikipedia.org/w/index.php?title=Tic-tac-toe&oldid=1219953149`, [Online; accessed 22-April-2024] (2024).

[4] Wikipedia contributors, Reversi — Wikipedia, the free encyclopedia, `https://en.wikipedia.org/w/index.php?title=Reversi&oldid=1208794846`, [Online; accessed 27-March-2024] (2024).

[5] L. V. Allis, Searching for solutions in games and artificial intelligence (1994).

[6] H. Takizawa, Othello is solved, arXiv preprint arXiv:2310.19387 (2023).