Project – IASD 2023
Monte Carlo methods for Othello

Alexis Georgiou

1 Introduction

Monte Carlo methods have been widely used in various fields to estimate complex systems and optimize decision-making processes. An interesting application of these methods is in the game of Othello, also known as Reversi. Othello is a two-player board game played on an 8x8 grid with 64 identical pieces, which are black on one side and white on the other. The game is won by the player who has more pieces of their color on the board when no more moves can be made. The pieces are be flipped to their other side depending on the move on every turn.
Monte Carlo methods are a powerful tool for analyzing the games. These methods allow us to simulate a large number of possible moves and outcomes, which can help players make informed decisions about their next move. By repeatedly simulating the game from the current state, Monte Carlo methods can be used to estimate the likelihood of winning and find the best move to make given the current state and a calculation limit.
In this project, we implemented the logic of the game and some of the methods explored in class. We also compare the performance of them for the game selected, by simulating a round robin tournament.

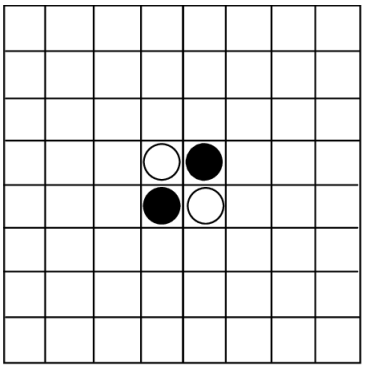

Figure 1: Othello's starting position

2 Monte Carlo Methods

2.1 Flat Monte Carlo

The flat Monte Carlo (flatMC) method starts by exploring each possible move equally and performs an equal amount of playouts for each move. For each move, the method keeps track of the number of playouts starting with that move and the number of playouts starting with that move that have been won. After all playouts are finished, the method simply selects the move with the greatest mean as the best move.

2.2 Upper Confidence Bound

The Upper Confidence Bound (UCB) algorithm is a variation of Monte Carlo Tree Search that aims to balance between exploration and exploitation of the game tree. It does this by choosing moves based on a score formula, that takes into account the average score of each move and the number of times it has been visited. The move with the highest UCB score is chosen before each playout. After performing a specified number of playouts, the algorithm returns the move that has been visited the most. This approach helps to prioritize the exploration of promising branches of the game tree and leads to more efficient search for the best move.

UCB score: $\frac{w_i}{n_i} + c\sqrt{\frac{\ln t}{n_i}}$ where

- w_i: number of wins after the i-th move
- n_i: number of simulations after i-th move
- c: exploration parameter
- t: total number of simulations of parent node

2.3 Upper Confidence Bound applied to Trees

This method builds a search tree of possible moves, and at each node of the tree, it uses the UCB formula to determine which child node to explore next. This formula balances between the exploitation of nodes that have been found to have high win rates and the exploration of nodes that have not been fully explored. The algorithm keeps track of the number of times each move has been played, the number of times each move has led to a win, and the total number of playouts. At the end, the move that has been played the most times in the search tree is selected.

2.4 Sequential Halving

This method is based on the idea of repeatedly splitting the set of moves into two halves and selecting the better half. The better half is retained for the next iteration, until only one move is left. At each iteration, the algorithm runs UCT (Upper Confidence Bound applied to Trees) simulations to estimate the expected reward of each move. The half with the biggest expected

reward is the best half. Naturally, the number of iterations required by Sequential Halving is proportional to the logarithm of the total number of legal moves, and the algorithm is generally more efficient than normal UCT in terms of speed and performance.

3 Results

We run a Round Robin tournament between the 4 Monte Carlo methods previously presented. We also added a simple method as a baseline that selects a move at random out of the legal available moves. In the results, we have a tuple of two numbers on each cell, the first number is representing the wins of the row method against the column method, the second number is representing the draws on the same match ups. We run a total of 100 games for every match up.
For example, $95, 0$ means that $flatMC$ won 95 games against $random\ move$, drawed 0 games and lost 5 games out of 100 total games.
All the methods run with $n = 100$ playouts limit.

	random_move	flatMC	UCB	BestMoveUCT	SequentialHalving
random_move		[5, 0]	[20, 1]	[3, 1]	[6, 1]
flatMC	[95, 0]		[63, 0]	[25, 2]	[10, 1]
UCB	[80, 1]	[37, 0]		[27, 0]	[7, 0]
BestMoveUCT	[97, 1]	[75, 2]	[73, 0]		[6, 0]
SequentialHalving	[94, 1]	[90, 1]	[93, 0]	[94, 0]	

Table 1: Round Robin results 100 games per match up, 100 playouts for every method

	random_move	flatMC	UCB	BestMoveUCT	SequentialHalving
random_move		[0, 0]	[1, 0]	[0, 0]	[2, 0]
flatMC	[10, 0]		[9, 0]	[2, 0]	[1, 0]
UCB	[9, 0]	[1, 0]		[1, 0]	[1, 0]
BestMoveUCT	[10, 0]	[8, 0]	[9, 0]		[1, 0]
SequentialHalving	[8, 0]	[9, 0]	[9, 0]	[9, 0]	

Table 2: Round Robin results 10 games per match up, 1000 playouts for every method

We can estimate each method's performance on Othello by looking at the results. First, all the methods are winning random move, which is intuitively obvious. UCB is the worst against random move and it also loses from everyone else, which makes it a bad method for our problem. UCT wins both $flatMC$ and UCB and $SequentialHalving$ is winning all of them. Also $SequentialHalving$ is much faster than any other method, which makes it the best method we tried for our game, although it is an extension of UCT.
As we increase the number of playouts for every method we can see that UCB gets weaker in comparing with other methods.
Overall performance of the methods is $random << UCB < flatMC < UCT < SequentialHalving$

4 Bibliography

Hingston, Philip Masek, Martin. (2007). Experiments with Monte Carlo Othello. ECU Publications. 4059 - 4064. 10.1109/CEC.2007.4425000.
Project's code: [*]
Monte Carlo Cazenave's class: [*]