Othello README

David M. Everly Jr.

Drexel University

CS510: Introduction to Artificial Intelligence

Dr. Ehsan Khosroshahi

November 20, 2024

Othello README

The program is run via the shell script “run.sh” and accepts the following case-sensitive shell commands:

All commands must be provided in the format <player 1><player 2><depth limit (unless both player 1 and player 2 are ‘human’ or ‘random’)>

Players alternate turns until the game reaches a completed state

Player 1 has token of “O” and takes first turn

Player 2 has token of “X” and takes second turn

Player-types have the following characteristics:

“human” – prompts user to select the next move from legal moves during their turn

“random” – player selects a random move from legal moves during their turn

“minimax” – player uses a minimax algorithm at specified depth limit to select the most logical move

“alphabeta” – player uses a minimax algorithm with alpha-beta pruning to the specified depth limit to select the most logical move

“MCTS” – player uses monte-carlo tree search algorithm. By default, this takes an integer time limit in milliseconds and runs the algorithm until the time expires. If number of iterations is desired, rather than time limit, one need only change the constant variable “TIMER” to False at the top of the file in mcts.py.

Implementation Details

Heuristic Function

A custom heuristic function was created to more accurately describe desired game states. Instead of the previous implementation, which returned a count of ‘x’ and ‘o’ tokens, the heuristic function in Othello.py gives each board space a score based on the distance from the edge of the board. As pieces on the edge of the board are least likely to have a flanking move, these spots are more stable and contribute more significantly to the game’s end state. The most important space is the corner, which can’t be flanked and is a permanent selection; each corner was given a heuristic value of 6 points to reflect the importance of that position. The heuristic point distribution is, therefore;

6 44444 6

4 33333 4

4 32223 4

43211234

4 32223 4

4 33333 4

6 44444 6

MCTS Modified Game

MCTS algorithm uses random players for playouts, but the game.playMCTS() function changes some details of the simulated games. Specifically, the games have an early termination heuristic intended to improve time complexity, which considers a game to be in a final state if the total score is outside the bounds of (-50,50). A simulated state which meets these criteria is terminated and the winner is chosen based on the current score. PlayMCTS also removes the print functionality to avoid the simulated games from printing to the terminal window by temporarily redirecting stdout to None, so the terminal window isn't cluttered with results of simulated games.

MCTS Timer

As mentioned above, mcts.py will implement a timed game if the constant TIMER = True, or a iterations-based algorithm is TIMER = False.

MCTS Reflex Moves

To improve time complexity and decrease the risk overlooking obvious moves, this implementation of MCTS will immediately take any corner move if it is available in the initial list of legal moves. Finally, considering that much of the runtime is spent looking up if a state has already been seen previously, a hash table was created with a determinate hash function to speed this process. The hash function could be improved with a randomization, but this was thought to be unnecessary for the current project and

likely to be quite time consuming since I had never attempted to create a solution involving a hash table before this project. There is also some residual code for a max heap, which I created but ultimately chose not to use as it did not improve runtime and added unnecessary complexity to the solution.

Results and Analysis

Original Implementation of Minimax with AlphaBeta Pruning

Table 1 depicts the win rate and median game time for 10 games against a randomized opponent from our previous implementation. This implementation uses the native scoring function.

Table 1. Win Rate and Median Game Time of Alpha Beta Minimax playing 10 Games against Randomized Opponent Using Native Scoring Using Various Depth Limits

Algorithm	**Depth Limit**	**Win Rate (%)**	**Median Time (seconds/game)**
Minimax + αβ	1	75	0.03
Minimax + αβ	2	85	0.05
Minimax + αβ	3	80	0.17
Minimax + αβ	4	65	0.53
Minimax + αβ	5	90	2.48
Minimax + αβ	6	100	7.54

Heuristic Method

Table 2 provides for comparison between the native scoring algorithm and the heuristic method, described above, for the minimax algorithm. While there was some minor improvement in win percentage, I would suspect that this is not statistically significant. The algorithm was run at depth of 7 for more accurate comparison to the MCTS algorithm, discussed below.

Table 2. Win Rate and Median Game Time of Alpha Beta Minimax playing 10 Games against Randomized Opponent Using Heuristic Scoring Using Various Depth Limits

Algorithm	Depth Limit	Win Rate (%)	Median Time (seconds/game)
Minimax + αβ	1	70	0.03
Minimax + αβ	2	80	0.05
Minimax + αβ	3	90	0.13
Minimax + αβ	4	90	0.63
Minimax + αβ	5	100	2.63
Minimax + αβ	6	100	15.22
Minimax + αβ	7	100	39.15

MCTS Performance

Table 2. Win Rate and Median Game Time of MCTS playing 10 Games against Randomized Opponent Using Heuristic Scoring Time or Simulation Iteration Limits

Algorithm	Time Limit (s/move)	Win Rate (%)	Median Time (seconds/game)
MCTS	1	100	23.46
MCTS	5	100	123.51
MCTS	10	100	254.17

Algorithm	Limit (simulations/move)	Win Rate (%)	Median Time (seconds/game)
MCTS	5000	100	223.86
MCTS	10,000	90	398.03

Discussion

Comparing the runtime of the MCTS algorithm with that of minimax with alpha-beta pruning, the implementation of MCTS performed best when given a shorter runtime of 1 second/move, with a median runtime of 23.46 seconds per game compared to the minimax with alpha-beta pruning at depth of 7, averaging 39.15 seconds per game. Given the relatively small state space of the game of Othello compared to other games such as Go, either algorithm was able to reach a reasonable efficiency with

high win rate. In games with a larger state space, minimax with alpha-beta pruning would likely become significantly less efficient, while MCTS would be expected to provide superior results. In this implementation of MCTS, I was surprised to find the overall time complexity to be unexpectedly large despite the addition of the hash table and shortcuts for taking obvious corner moves. I expected to find MCTS superior to minimax with alpha-beta pruning, but in practice, I found my minimax + alpha beta pruning to be superior in runtime complexity while providing similar win rates for randomized games.