Othello Analysis Report

Bradley Hampton

CS 580

Professor Yaohang Li

1 November 2024

1 Program Design

1.1 Data Structures

This program features three primary data structures:

- Grid, which stores and mutates the game state.
- Graphics, which renders a GUI for a player to see and interact with the game.
- Othello, which links together the Grid and Graphics, and passes control between the human player and the AI.

Othello has 4 parameters, which controls the scale of the game and how the AI works:

- n, which is the size of the square Grid
- algorithm, a function which controls how the AI will traverse or parse through potential game states
- ply, which controls how far the algorithm should look ahead
- heuristic, a function which evaluates a game state.

The AI itself only performs operations on the Grid, it does not ever access Graphics. The algorithm and heuristic are obtained imported from algorithms.py and heuristics.py respectfully. The program itself accepts command-line arguments for the user to select which arguments to run Othello with.

When the AI is prompted to make a move, it constructs a game-tree from scratch, without looking at a cache from previous moves. Future work could include implementing memorization, which could drastically improve the performance for higher look-ahead.

1.2 Heuristic Functions

1.2.1 Count

This heuristic function counts the chips of each color in a grid. Next, the difference between these two values is computed.

In this implementation, we always compute the difference as the heuristic score:

$$\text{heuristic\_score} = \text{number of white chips - number of black chips}$$

Which means that a higher difference maximizes having more white chips over black chips. All heuristic functions in this program accept the player we are attempting to maximize as an argument. Since this is a zero-sum game, if we find that we need to maximize for black (not white), then we will multiply by -1 to obtain our final heuristic score.

1.2.2 Frontiers

For each space in the grid, its number of frontiers is equal to the count of empty spaces adjacent or diagonal to itself. In other words, one space away in all cardinal and ordinal directions.

This heuristic function finds the difference in sums for each player's frontiers, attempting to maximize the opposing player having more frontiers.

$$\text{heuristic\_score} = \sum_{\text{chip}}^{\text{black}} \text{frontiers(chip)} - \sum_{\text{chip}}^{\text{white}} \text{frontiers(chip)}$$

Like in 1.2.1, given that this is a zero-sum game, the heuristic score is multiplied by -1 when maximizing instead for black.

If this heuristic evaluates a grid where the game is over, then it will return infinity if the max player would win, and negative infinity if the min player would win.

Otherwise, the heuristic score is returned.

1.2.3 Positions

This heuristic is similar to the function described in 1.2.1, except cells on the edge and corners of the grid are weighted. Edges of the grid are assigned a cell weight of $\text{floor}(n/2)$. Corners of the grid are assigned a cell weight of $\text{floor}(n/2)^2$. Cells that are neither corners nor edges are assigned a cell weight of 1.

$$\text{heuristic\_score} = \sum_{\text{chip}}^{\text{white}} \text{cell\_weight}_{\text{chip}} - \sum_{\text{chip}}^{\text{black}} \text{cell\_weight}_{\text{chip}}$$

Like in 1.2.1, given that this is a zero-sum game, the heuristic score is multiplied by -1 when maximizing instead for black.

If this heuristic evaluates a grid where the game is over, then it will return infinity if the max player would win, and negative infinity if the min player would win.

Otherwise, the heuristic score is returned.

2 Performance Statistics

Tables 1-3 show the results from a test on each algorithm and heuristic with 2-ply, 4-ply, and 6-ply look-ahead. For these tests, the human-player move was chosen arbitrarily. Tests involving 6-ply look-ahead is reserved for the Alpha-Beta algorithm, as the Minimax algorithm on 6-ply tests caused each move from the AI player to take several minutes.

Algorithm	Heuristic	Ply	Average Turn	Longest Turn
Minimax	Count	2	120ms	353ms
Minimax	Count	4	7sec	35sec
Alpha-Beta	Count	2	14ms	28ms
Alpha-Beta	Count	4	203ms	881ms
Alpha-Beta	Count	6	6.8sec	22sec

Table 1: Performance Statistics on the Pieces Heuristic.

Algorithm	Heuristic	Ply	Average Turn	Longest Turn
Minimax	Frontier	2	183ms	625ms
Minimax	Frontier	4	6.8sec	23sec
Alpha-Beta	Frontier	2	15ms	31ms
Alpha-Beta	Frontier	4	552ms	1.5sec
Alpha-Beta	Frontier	6	11sec	29sec

Table 2: Performance Statistics on the Frontier Heuristic.

Algorithm	Heuristic	Ply	Average Turn	Longest Turn
Minimax	Positions	2	134ms	475ms
Minimax	Positions	4	26sec	134sec
Alpha-Beta	Positions	2	16ms	35sec
Alpha-Beta	Positions	4	441ms	1.3sec
Alpha-Beta	Positions	6	21sec	74sec

Table 3: Performance Statistics on the Positions Heuristic.

3 Performance Analysis

3.1 Mini-Max Algorithm

The Mini-Max algorithm scales poorly, so tests were only run with up to 4-ply. Since Mini-Max effectively limits the range of look-ahead in this way, the skill of an AI using this algorithm is diminished, compared to the Alpha-Beta algorithm.

3.2 Alpha-Beta Pruning Algorithm

Alpha-Beta Pruning results in better time efficiency compared to the Mini-Max Algorithm. As a result, it was reasonable to test the alpha-beta algorithm on with larger plys than the Mini-Max algorithm. Generally, running a Heuristic on 6-ply on Alpha-Beta has better execution times than running the same heuristic on 4-ply with Minimax (except for with the Frontiers heuristic).

When playing against both algorithms with otherwise identical arguments, I did not observe any significant difference in the playing skill, as should be guaranteed by the alpha-beta pruning algorithm.

These implementations of Alpha-Beta pruning and Minimax do, however, tend to make different decisions from each other. Both algorithms side-by-side, using otherwise identical settings, with the human-player making an identical sequence of moves in each game.

When playing against the same algorithm with exact identical settings side-by-side, with the human player making an identical sequence of moves in each game – the AI responded with the same moves in each game.

When repeating this experiment, with each game being run with different algorithms, it was observed that the AI would occasionally respond with different moves in each game. I believe that this is due to the differences in how each algorithm handles cases where multiple moves have the same heuristic scores.

In alpha-beta pruning, a branch is pruned if its heuristic score is less than or equal to the best heuristic score that was found in a subtree that was traversed earlier.

In the Mini-Max algorithm, the program applies Python's built-in max function to the set of heuristic scores returned by all subtrees. This means that the move that would be chosen in the event of a tie is effectively randomized (since sets are in no guaranteed order).

3.3 Count Heuristic

The most intuitive but also the weakest heuristic. Provides the best execution performance out of the heuristics, most likely because it requires the least amount of overhead.

The most trivial to beat as it doesn't see long-term plays, or understand the positional advantage of certain cells, this is especially damaging when the ply is small.

Generally, I was able to beat this heuristic most of the time just by prioritizing the corners. It is especially easy to obtain an edge or corner when having it cannot immediately put the player into a position to flip several opponents' chips in the next move.

3.4 Frontier Heuristic

Frontiers tend to have faster execution times than Positions, even though it should have the greatest amount of overhead. The heuristic itself is more intensive and performs more checks and calculations compared to Count and Positions; however, it saves time in the end by reducing the search space of the game tree. Compared to Positions, each test run of Frontier appears to take about half the time as other Heuristics when all other settings are the same.

By attempting to minimize its number of frontiers, the AI effectively attempts to reduce the amount of potential moves the human player could respond with. By making moves in this way, the AI on average will need to evaluate less complex game trees, reducing the execution time.

This heuristic does not seem to result in the best playing ability against a human-player. Corners and edges do not appear to be prioritized by the algorithm heavily enough. If a human-player obtains a corner, it may result in them obtaining a large amount of chips that increases the count of their frontiers. In some ways, this heuristic may incentivize moves those results in the AI losing chips, because it would result in the human-player opening more frontiers.

I expect that this heuristic could be useful to reduce to the size of a game-tree so that the algorithm would more reasonably be able to look further ahead (with alpha-beta) than when combined with other heuristics, but it does not make the best moves on its own and is easily "tricked."

3.5 Positions Heuristic

This heuristic has relatively minimal overhead compared to Frontiers but tends to take the longest to make moves for the AI player.

I was unable to beat this heuristic on average when the ply was equal or greater than 4. Out of all heuristics, I believe that this one makes the best decisions that puts itself at an advantage. Obtaining a key edge or corner tile in the short-term gives the AI tempo, which it can use later into the game to secure a win.

If I were to work on this program for longer, I would likely use the positions heuristic as a starting point but reduce the weight of the edge or corner cells as the game gets closer to its conclusion. In Othello, I see taking a corner or edge tile as equivalent to either losing a queen or rook in chess. It doesn't automatically mean that the game is a loss, but it makes it much more difficult to return to an advantageous position.