# On Othello and Parallelism: Constructing a Simple AI

github.com/joshfeds/OthelloParallelism


Jarod Davies
*Undergrad Student, Dept. Computer Science*
*University of Central Florida*
Orlando, FL, USA
ja371706@ucf.edu

Anna MacInnis
*Undergrad Student, Dept. Computer Science*
*University of Central Florida*
Orlando, FL, USA
an504194@ucf.edu

Joshua Federman
*Undergrad Student, Dept. Computer Science*
*University of Central Florida*
Orlando, FL, USA
jo239650@ucf.edu

Nicholas Rolland
*Undergrad Student, Dept. Computer Science*
*University of Central Florida*
Orlando, FL, USA
ni213570@ucf.edu



***Abstract*—Othello, also known as Reversi, is a tile-based board game, where the goal is to capture as many pieces on the board as possible. Othello has simpler rules than more complex games, e.g., Chess, but still presents a massive $10^{58}$ possible board positions, on a traditional 8×8 board. Thus, an AI to play Othello effectively can be constructed using methods found in similar board games.**

**In this paper, we parallelize a well-known algorithm for playing board games like Othello optimally, specifically the Minimax algorithm with alpha-beta pruning, which is widely used for searching game trees of other board games, like Chess. The Minimax algorithm searches the tree of next possible game states and aims to minimize loss in the worst case possible for a given move, while alpha-beta pruning skips moves that need not be considered.**



***Index Terms*—Alpha-beta pruning, Minimax, Othello AI, Parallel algorithms**


## I. Introduction

### A. Othello Rules

Othello is a fairly simple two-player game that takes place on a square n×n board, often 8×8. Typically, one player uses black pieces and the other player uses white pieces. The initial state of the board is as follows in Fig. 1, with white to move indicated by the four translucent rings.

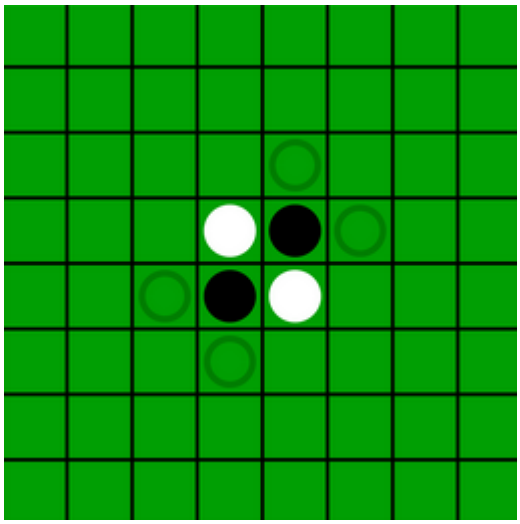

Fig. 1. The starting position of a game of Othello.

The historical version of the game known as Reversi starts with an empty board, instead of the layout above. The basic rules are as follows:

- Players take turns placing a piece of their color on an empty cell. A player may only place their piece on a cell if it "sandwiches" at least one of the opponent's pieces between the new piece and an existing piece on the board, in at least one of the 8 compass directions; that is, the new piece follows one or more pieces of the opposite player's color, and then a piece of the same color [1].
- Then, the player effectively "steals" the sandwiched pieces; that is, all pieces between the bordering pieces are changed to the current player's color [1].
- For example, in Fig. 1 above, the rings indicate white's valid moves, since the new piece would trap a black piece between itself and an existing white piece.
- If a player cannot make a move, the turn is automatically passed to the opponent. The game ends when either all cells are filled or neither player can make a move [1]. The player with the most pieces on the board wins the game.

## II. Implementation

### A. Implementation of Game Logic

The Board class simulates the main functionality of an Othello game. It contains fields representing the board state itself as a 2D integer array, the current player, and a global mapping of possible valid moves with associated data to assist in efficiently modifying the board state to make the move. The Board class encapsulates all the functionality necessary to track valid moves the current player may make, modify the board state and other fields to simulate making a move, and display the current board and other game information for debugging purposes.

### B. Minimax Decision Tree

The Minimax game tree for Othello contains nodes which represent possible moves a player may make at varying states in a valid Othello game. Each node represents a move and contains a number of fields defining information about that move: the player making the designated move and a score that

represents how advantageous that move is for that player. A given node also stores a copy of its associated board state– the state upon which the node's player may make the associated move. Any game tree is initialized with one of four roots: one for each possible move the player may make on the initial board pictured above. The children of any given node represent the moves that the node's opponent may make after the node's move occurs.

The initial approach to implementing this structure was to construct an entire completed tree initially. However, this approach presented significant runtime and memory availability issues since the constructor recursively built subtrees representing every possible Othello game. The algorithm has since been modified to build the game tree on an as-needed basis to only construct nodes that the AI must check.

### C. Frontend Display

The graphics for our remake of Othello were written using JavaFX, a GUI library for writing desktop applications in Java. The window which displays everything is a Stage object, which can display one or more Scene objects, which are containers for different visual elements. The Board scene chiefly contains a BorderPane object, which displays a central box element, with optional panels to the left, right, top, and bottom.

The main game board is contained in the center element of the BorderPane, and is made of green rectangles with black outlines to form the grid. Using basic JavaFX Circles and Rectangles, It is drawn by looping through the array representation of the current board state: if there is no piece, simply draw the green cell. If there is a piece, draw the green cell, then a black or white piece on top. Finally, possible moves on the human player's turn are indicated with transparent rings. The ring darkens when hovered over with the mouse, using setOnMouseEntered and setOnMouseExited events. All of these are put inside a Group, a simple container for shapes.

Other elements on the Board scene include the current number of each player's pieces, as well as some player avatar images. These are all organized within VBoxes, containers which arrange their elements vertically. The left and right panels of the BorderPane each contain one VBox with a brown background: left to show the human player's info, and right to show the AI's info. The entire Board scene is shown in Fig. 2. To make the game feel more complete, there are two more scenes, Main Menu and About, which contain buttons to link all the scenes together.

### D. Frontend Integration

To combine the frontend and backend of the program, three main classes are used; BoardDrawer, Board, and Minimax. BoardDrawer handles the creation and drawing of the board for every move from both players, as well as receiving mouse input from the human player. Board holds the game board for a given game. Minimax handles the game tree creation and all possible branches off the tree. Initially, we get a HashSet of all possible moves (variable name: nextMoves) that could be

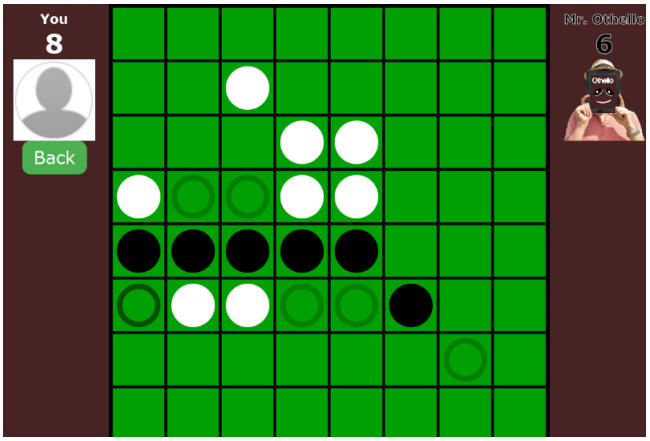


Fig. 2. The Board Scene, displaying the board state and player scores partway through a game of Parathello.

made in a given turn. Once a player clicks on a move, it is checked against nextMoves for validity. If it is a valid move, we go through the roots of the game tree to check for the correct node. Once it is found, we move down the tree to the next set of leaves, updating the game tree's board accordingly. Next, we retrieve the best move found within the game tree and repeat the same process of making a move for the AI. A visualization of this process can be seen in Fig. 3.

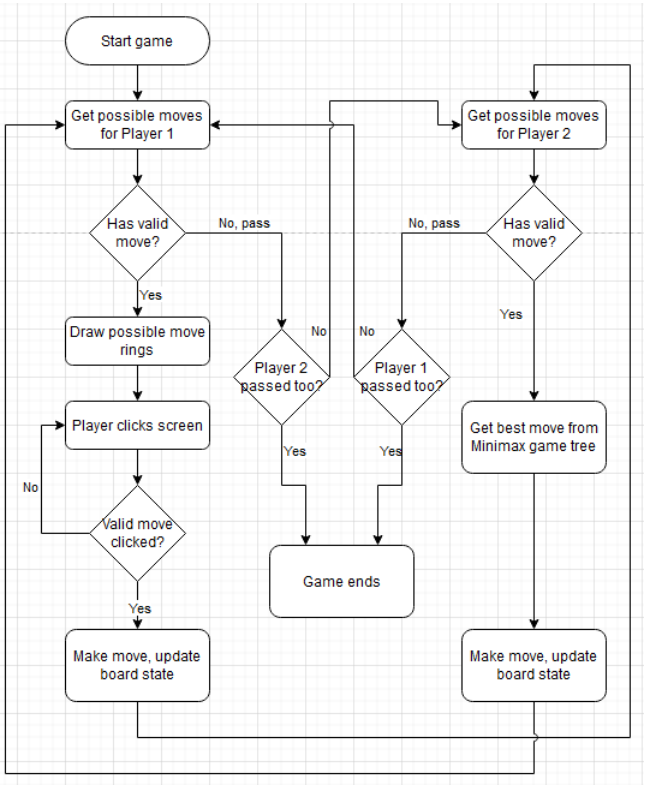


Fig. 3. Flowchart of game progression between human and AI player in class BoardDrawer.

The column of nodes on the left outlines the actions for

processing input for the human player. Once a move is clicked and made, control passes to the column of nodes on the right, allowing the AI to make its move. If either player has no moves available, then the turn is passed to the other player; but if they cannot play either, the game ends immediately. This typically happens when the board completely fills up, but can happen earlier as well with empty cells appearing in the final board state.

## III. SCORE CALCULATION AND DECISION-MAKING

As the AI plays Othello, it encounters a recurring task. Given a list of moves, represented by nodes, it must decide which move to make to maximize its chances of winning, or at least to tip the scales in its favor. Move scoring is a vital component dictating how the AI determines which move to make at any given game state. Each node in the current tree is given a score, which determines the AI's next move. Calculating and assigning these scores is a process that happens in three key steps.

### *A. Build Lookahead*

As mentioned previously, the game tree is constructed on an as-needed basis due to the limited nature of memory and the unrealistic computation demands associated with constructing an entire, complete tree. This means that before node scores can be determined and evaluated, the portion of the tree representing the current scenario as well as possible future scenarios must be constructed.

Due to limitations in both memory and runtime, it is not feasible to construct completed subtrees that reach all the way to end-game states for each node in a given list of possible moves unless the game has progressed to a point where possible end-game board states are well within reach. Due to this limitation, this algorithm constructs a certain, predetermined amount of levels below the current level for each decision it must make. This constant lookahead value can also be conceptualized as the amount of moves in the future the AI is able to predict and account for. The decision between lower versus higher lookahead represents a tradeoff between efficiently utilizing resources and the amount of possible future scenarios the AI is able to anticipate. This implementation uses five levels of lookahead, which provides a sufficient balance between these conflicting concerns.

### *B. Calculate Scores for Leaves*

After subtrees for each node have been constructed, scores for leaf nodes are determined based on types and amount of board locations owned by the AI and human on the associated board state. Locations on the board can either be beneficial or harmful to the AI's chances of winning. Below are the types of locations and their associated value:

- Corner squares are the most beneficial to obtain since they can never be stolen by the opponent [3]. These are a cornerstone for many late-game situations. If the AI owns a corner, the overall score is incremented by a large positive value. If the human owns a corner, the score is instead decremented by this value.
- Edge squares are also valuable since they are protected by their adjacent border [3]. AI-owned edges cause a substantial score increment while human-owned edges cause the score to decrease by that same amount.
- Buffer squares, or squares which immediately surround a corner, are beneficial if their associated corner is owned by the current player [3]. They are harmful otherwise since taking them can present the opponent with a chance to obtain their corner [3]. The score of a node increases for each helpful buffer owned by the AI or unhelpful buffer owned by the human. The score decreases for each harmful buffer owned by the AI or helpful buffer owned by the human.
- Interior squares are surrounded by non-empty locations on all sides. These are somewhat beneficial since they are not as easy for the opponent to steal as other pieces [3]. AI-owned interior squares increment the score while human-owned interior squares decrement the score.
- Frontier squares have at least one empty square in their perimeter. These are somewhat harmful since they are more easily stolen [3]. AI-owned frontier squares decrement the score while human-owned frontier squares increment the score.

To assign scores to leaf nodes, the algorithm considers each taken square on the associated board state at that point in time. A leaf's score is either incremented or decremented based on each of the types of locations obtained. Some locations carry more weight than others. For instance, corners modify the overall node score by twenty, edges change it by ten, and interior nodes only change it by one.

Note that the designated value constant of some type of board location is either added to or subtracted from the overall score depending on whether the location is beneficial or harmful and whether the location is owned by the AI or the human player. Scores of leaves are based on how optimal the associated board state is for the AI. As such, beneficial locations owned by the AI improve the score while that same location owned by the human reduces the score by the same amount. The same is true for suboptimal locations: they represent a score decrease when owned by the AI and a score increase when owned by the human player.

This approach allows the score to most accurately measure the quality of the entire board state so that it may be compared against other board states associated with other nodes.

### *C. Minimax Algorithm*

This implementation uses the Minimax algorithm to determine scores for the leaves' ancestors. Each node is designated as either max– the AI, or min– the human player. Max nodes wish to follow a path that maximizes the score [4]. Min nodes wish for the opposite: to follow a path that minimizes the score [4]. In accordance with these wishes, a max node will adopt the highest score from its children, and a min node will adopt the lowest score from its children. This idea makes sense in the

flow of a zero-sum two-player game such as this. If one player can predict that making a certain move opens an opportunity for the opponent to claim a large amount of squares, the player is less likely to make that move. The logic of Minimax follows this method of strategizing and allows the AI to predict future scenarios and make an informed decision.

Minimax allows the AI to analyze potential sequences of moves and select the one that leads to the most favorable outcome. It's built on the assumption that the human player will do the opposite: select the least disadvantageous outcome [4].

### D. Decision Making

After the Minimax algorithm successfully propagates scores up the length of the current tree, what remains is a simple matter of choosing the node, or move, with the largest score. The AI then makes this move before waiting for the human player to make their next move, and the process repeats until the game concludes.

## IV. Alpha-Beta Pruning

Alpha-Beta Pruning is an optimization to accompany the Minimax algorithm [2]. It optimizes the algorithm by removing branches from the tree once it has been determined a better branch/move already exists [2]. For the purposes of Othello, Alpha-Beta Pruning would allow the AI to prune subtrees that aren't valuable by removing the associated children. For instance, if a maximum node already has a value of 3, then a minimum node below it finding a value of 2 means it doesn't matter how many more values less than 2 it finds towards its minimum, since this branch will no longer be picked at all by that 3 node above it. If an instance of this happens close to the root of the tree, as can commonly happen in Othello, it helps the runtime of the AI immensely. Furthermore, this implementation can work well with some forms of concurrency. This project uses concurrency in a way which when combined with Alpha-Beta Pruning will have a large effect on how quickly the program can calculate the best option.

## V. Concurrency

### A. Primary Focus

The three-step process the AI undertakes to select a move from a list of nodes is by far the most time-consuming and complicated process in this implementation. It involves multiple depth-first traversals, which are $O(N + E)$ where $N$ is the amount of nodes and $E$ is the amount of edges connecting those nodes. As such, this process was the focus of parallelization.

### B. Resources

Classes from Java's concurrent package were used to parallelize this algorithm. An ExecutorService was used to manage the thread pool and assigned tasks.

### C. Delegating Tasks

Subtrees must first be constructed for each node in some list of nodes representing the AI's choices. Constructing lookahead branches for some single node in the AI's list of choices is considered a single task that can be assigned to a thread in the thread pool. Siblings' subtrees are not connected or related to each other by any means, so they are safe for threads to access in parallel.

Calculating node scores, similarly, involves depth-first traversals over each subtree of each node. This task is broken down into a traversal for each node and distributed among the threads in the same method as illustrated previously.

### D. Avoiding Race Conditions

Special care was taken to account for race conditions that could occur if one thread starts trying to assign scores before another finishes building the associated subtree. Consider the following scenario: Some thread $A$ is currently calculating scores for the nodes in the subtree originating from some node $N$. However, some other thread $B$ has not finished constructing this subtree. Thread $A$ might then mistakenly treat the last nodes constructed by thread $B$ as leaf nodes. This results in thread $A$ ignoring a portion of $N$'s subtree, which defeats the purpose of using a predetermined lookahead value.

Another opportunity that threatens race conditions is the final decision the AI makes by iterating over the list of nodes and choosing the one with the highest score. If this process begins before all scores have been appropriately assigned, the wrong node may be selected or an exception may occur since score values are initially null.

To avoid this scenario, threads use two CountDownLatches from Java's concurrent package to communicate with the main thread responsible for delegating thread tasks and selecting the best move. A CountDownLatch is a thread-safe shared counter that can be decremented by some thread when it finishes a task. Two separate latches are each initialized to hold a value equal to the amount of nodes, or moves, for the bot to consider.

First, the main thread assigns all subtree construction tasks to the thread pool, then waits for one of the latches, the "build latch" to signify that all threads are finished. A thread responsible for building a subtree only decrements the latch when it is finished with its current task. The main thread waits until the value in the latch reaches zero before assigning score calculation tasks. The score calculation process is handled very similarly. The other latch, known as the "score latch" signifies to the main thread when all threads are finished calculating all scores. The main thread waits for this signal before determining the node with the maximum score.

## VI. Results

### A. Execution Time

Determining the amount of threads to use in the thread pool involved some experimentation. To do this, the getBestOption method, which takes a list of nodes and returns the node with the highest score and is the focus of parallelization, was tested on four different board states. The board states had four, eight,

eleven, and fourteen options to consider respectively. Various amounts of threads were tested, ranging from three to four to all other multiples of four up to and including forty. A purely sequential version was also tested. Each thread amount, including zero, was tested 100 times for each board state, and runtimes were averaged across each trial. Fig. 4 displays these average times in nanoseconds.

| | Test States (How many moves to evaluate) | | | |
|---|---|---|---|---|
| Threads | 4 Nodes | 8 Nodes | 11 Nodes | 14 Nodes |
| 0 | 4.12E+06 | 2.60E+08 | 2.00E+08 | 1.71E+09 |
| 3 | 3.48E+06 | 9.75E+07 | 7.31E+07 | 9.69E+08 |
| 4 | 2.29E+06 | 9.14E+07 | 7.13E+07 | 1.06E+09 |
| 8 | 2.18E+06 | 9.91E+07 | 7.71E+07 | 1.13E+09 |
| 12 | 2.16E+06 | 9.90E+07 | 7.76E+07 | 1.26E+09 |
| 16 | 2.07E+06 | 9.98E+07 | 7.99E+07 | 1.21E+09 |
| 20 | 1.97E+06 | 1.01E+08 | 7.89E+07 | 1.21E+09 |
| 24 | 1.94E+06 | 1.03E+08 | 7.81E+07 | 1.17E+09 |
| 28 | 1.96E+06 | 1.02E+08 | 7.75E+07 | 1.23E+09 |
| 32 | 1.98E+06 | 1.02E+08 | 7.85E+07 | 1.19E+09 |
| 36 | 1.91E+06 | 1.03E+08 | 8.03E+07 | 1.30E+09 |
| 40 | 1.98E+06 | 1.03E+08 | 7.97E+07 | 1.30E+09 |
| Execution time represented in nanoseconds | | | | |

Fig. 4. Average execution times of deciding the best move for various sized thread pools on various board states.

From the data above, it is apparent that four threads is consistently fast even in varying scenarios. Four threads do not always perform the fastest, but often perform at least faster than most other thread pool size options. Four threads also demonstrate impressive differences in execution times when compared with zero threads.

- For the board with four possible moves, four threads conclude the decision making task in about 55.5% of the time it takes the sequential implementation.
- On the board with eight possible moves, four threads conclude the task in about 35.2% of the time it takes the sequential implementation.
- On the board with eleven possible moves, four threads finish in about 35.7% of the time it takes for the sequential implementation.
- On the board with fourteen possible moves, four threads finish in about 62% of the time it takes for the sequential implementation.

Overall, after taking the average of each of these percentages, four threads take about 47% of the time it takes for a sequential implementation to complete the same decision-making task. This means that the concurrent approach is, in many cases, more than twice as fast as the sequential approach.

### *B. Performance against Humans*

To test the AI's ability to make the correct choices, two players each competed against the bot for ten games, adjusting the distance the bot would construct and consider within the game tree. Detailed results of these 20 games can be seen in Fig. 5. Between both players, the bot went 12-8, maintaining a 60% win rate. As the bot was able to look further ahead, it was able to have greater success in winning the game. In eight games where the player played against the bot that could look five steps ahead, only two were won. However, performance begins to degrade slightly after six steps, with a noticeable delay of up to a second on the AI's turn, and the game is not always playable for a full game at seven steps or greater depending on hardware constraints.

| Lookahead | Josh | AI | Nick | AI |
|---|---|---|---|---|
| 3 | 30 | 34 | 19 | 45 |
| | 14 | 50 | 38 | 26 |
| 4 | 30 | 34 | 36 | 28 |
| | 28 | 36 | 35 | 29 |
| | 51 | 13 | 36 | 28 |
| | 50 | 14 | 29 | 35 |
| 5 | 41 | 23 | 19 | 45 |
| | 21 | 43 | 26 | 38 |
| | 41 | 23 | 30 | 34 |
| | 31 | 33 | 26 | 38 |

Fig. 5. Results of 20 total games played against the AI by two human players, with varying lookahead values for the AI. Blue cells indicate an AI victory for that game, and green cells a human victory.

## VII. Conclusion

The Othello AI is capable of not only playing against a human in the board game Othello, but also making informed decisions by predicting future game states. Notable features include the game logic necessary to simulate turn-based Othello, backend to frontend communication, Minimax game tree construction, node score calculation using board position value and the Minimax algorithm, and optimization resulting from alpha-beta pruning and multi-threading. Further research could be conducted to hone the constants used to assign specific values to types of board positions in the hopes of further improving the AI's win rate. With the use of parallel processing, the AI is able to more quickly decide which move provides itself with higher odds for victory.

## References


[1] Worldothello.org. "Official Rules for the Game Othello." Available: https://www.worldothello.org/about/about-othello/othello-rules/official-rules/english, [Accessed: Mar. 4, 2024].

[2] JavaTpoint.com. "Alpha-Beta Pruning." Available: https://www.javatpoint.com/ai-alpha-beta-pruning, [Accessed: Mar. 7, 2024].

[3] K. Galli, How Does a Board Game AI Work (Connect 4, Othello, Chess, Checkers) - Minimax Algorithm Explained. (2019). Accessed: Mar. 8, 2024. [Online video]. Available: https://www.youtube.com/watch?v=y7AKtWGOPAE

[4] Baeldung.com. "Introduction to Minimax Algorithm with a Java Implementation." Available: https://www.baeldung.com/java-Minimax-algorithm, Jan. 11, 2024. [Accessed: Mar. 10, 2024].