# COMP 424 Final Project Game: *Reversi Othello* Report

This document discusses the implementation and performance evaluation of a Reversi Othello-playing AI agent.

## 1. Best Approach: Advanced Heuristic-Driven Minimax Algorithm with Alpha-Beta Pruning

The strongest algorithm implemented for playing Reversi Othello leverages a combination of iterative deepening, minimax-based search, alpha-beta pruning, and heuristic evaluation. This approach balances computational efficiency and strategic depth by pruning unpromising branches in the search tree while maximizing the depth of exploration within available time constraints. Iterative deepening ensures that the search produces the best possible move within a time limit, progressively refining decisions as computation allows. The evaluation function, based on domain-specific heuristics like mobility, corner control, and stability, plays a pivotal role in assessing board positions and guiding decision-making. The algorithm achieves strong performance, offering competitive play quality through its strategic foresight and tactical accuracy. These conclusions are drawn from iterative design and empirical testing, which consistently showed this method outperforming simpler algorithms and heuristics in both speed and accuracy of play.

## 2. Agent Design

The AI agent for playing Reversi Othello employs a minimax algorithm with alpha-beta pruning, supplemented by heuristic evaluation to determine optimal moves within a restricted time frame. This design balances computational efficiency with effective decision-making under the constraints of the game.

### 2.1 General Theoretical Elements

**Minimax Algorithm:** The minimax algorithm is a decision rule commonly used in zero-sum games to minimize the potential maximum loss while maximizing potential gain. It recursively explores the game tree, evaluating each possible move up to a certain depth and assigning a score based on the resulting game states. The agent alternates between “maximizing” (its own turns) and “minimizing” (opponent’s turns) to simulate adversarial play [2].

**Alpha-Beta Pruning:** To optimize the minimax algorithm, alpha-beta pruning reduces the number of nodes evaluated in the game tree by eliminating branches that cannot influence the final decision. Alpha ($\alpha$) represents the best score achievable by the maximizer, while Beta ($\beta$) represents the best score achievable by the minimizer. If a node’s evaluation falls outside the range defined by $\alpha$ and $\beta$, further exploration of that branch is unnecessary [2].

### 2.2 Specific Implementation Details

The agent is implemented as a Python class `StudentAgent`, which inherits from a base `Agent` class. It uses helper functions from `helpers.py` such as `get_valid_moves` to determine legal moves and `execute_move` to simulate board changes. The main decision-making function is `step`, which identifies the optimal move based on the heuristics described below.

#### Heuristic Evaluation Function

The heuristic function evaluates the desirability of a board state using several weighted factors. These factors balance immediate gains against long-term positional advantages:

1. **Corner Control:** The corner control heuristic evaluates the player's dominance over the corners and the associated C-squares and X-squares. C-squares are the squares adjacent to the corners, while X-squares are the diagonal squares next to the C-squares [1]. The heuristic also considers the stability of these positions by rewarding players for controlling the squares adjacent to the corners. This added layer of stability enhances the evaluation by acknowledging the importance of securing surrounding squares. The function penalizes the player if a corner is neutral or controlled by the opponent, as occupying nearby squares without controlling the corner itself can result in a potential corner takeover by the opponent.

2. **Mobility:** This heuristic measures the mobility of both players, based on the number of legal moves available. The difference between the player's and opponent's mobility is then normalized to a percentage.

3. **Piece Count:** This heuristic evaluates the difference in piece count (the number of pieces on the board) between the player and the opponent, relative to the total number of squares.

4. **Capture Potential:** It calculates the number of opponent pieces that can be flipped by a specific move. This value is computed by simulating the move and counting all affected tiles.

5. **Edge Control with Wedging:** This heuristic evaluates edge control and the potential creation of wedges. A wedge occurs when there are an odd number of empty squares between two pieces of the same color along an edge [1]. In such a case, the opponent can potentially be forced into a position where they have no choice but to play into a vulnerable position, allowing the player to trap their pieces. This heuristic rewards players for controlling edge squares and penalizes for the opponent's control.

6. **Stability and Dynamic Weights:** The agent's design focuses on evaluating the stability of each disc on the board, categorizing them as super stable, stable, semi-stable, or unstable based on their position and surrounding discs.

   The `initialize_static_weights` function assigns different values to positions like corners, near-corners, edges, and neutral squares, helping the agent prioritize important areas of the board, such as corners, which are highly valuable in the game. The `calculate_stability` function, along with its helper functions (`get_disk_stability`, is_near_corner, and is_semi_stable), determines the stability of each disc, giving the agent insight into how secure each disc is. Finally, the calculate_dynamic_weights function combines these stability values with the static weights to create dynamic weights for each position. This helps the agent evaluate the current state of the board more effectively, guiding its decision-making process by scoring board positions based on both their value and the stability of the discs. Together, these functions form the heart of the agent's heuristic evaluation, allowing it to assess which board positions are most favorable at different stages of the game.

The importance of heuristics varies across the game phases, as reflected in their weighted adjustments in the evaluate_board function. The game is divided into three phases—early, middle, and late—based on the number of pieces on the board. The relative importance (weight) of each heuristic in calculating the final evaluation score depends on the game phase and evolves as the game progresses. This ensures the agent adapts its strategy to different phases, enhancing its overall performance in diverse scenarios.

In the early game, mobility is given a very high weight (2.0), reflecting its critical role in maintaining flexibility, while corner control and piece count are weighted lower (0.7 and 0.5, respectively). Edge control receives a moderate weight (1.0), balancing its positional significance without compromising mobility. In the middle game, priorities shift to balance factors: corner control and edge control are highly weighted (3.0 each), emphasizing strategic positioning, while mobility (1.5) remains significant but slightly reduced. Piece count and count capture are given moderate-high importance (1.3 each), reflecting their growing influence. In the late game, corner control and edge control gain dramatic importance, with very high weights (8.0 and 7.0, respectively), as these factors often determine the game's outcome. Piece count and count capture also increase in importance (2.0 each), while mobility is de-emphasized (0.7), reflecting its reduced relevance in final positions.

It's worth noting that the weights for corner control and edge control are on a different scale compared to the other three heuristics because mobility, piece count, and count capture are normalized to account for board size and other factors. This discrepancy explains why the weights for corner and edge control appear much larger. These weights were fine-tuned through testing and adjusted based on their impact on gameplay. However, there's room for improvement. More extensive and conclusive testing, such as running additional automated games and simulations, could refine these weights further and enhance the evaluation's effectiveness.

Finally, all heuristic components are combined into a single final evaluation score. The dynamic board score, which represents stability, is included but not weighted separately, as its influence is typically reflected in other heuristics such as edge and corner control, even though stability can vary across phases.

### Move Ordering

The alpha-beta pruning function utilizes move ordering by sorting moves according to their heuristic evaluations. This approach enhances the chances of discarding suboptimal branches early, effectively shrinking the search space.

**Search Depth and Time Management**

To ensure compliance with the game's time constraints, the agent uses iterative deepening. It begins evaluating moves at a shallow depth and progressively deepens the search while monitoring elapsed time. The variable `time_limit` ensures the agent completes evaluations within the allotted 2-second window.

## 3. Quantitative Performance Analysis

### 3.1 Search Depth

**Achieved Depth:** The agent attempts to explore deeper search levels depending on available time. On average, it reaches a depth of 3 to 7 moves ahead in most games across board sizes.

**Explanation:** The agent uses iterative deepening with alpha-beta pruning, where the depth depends on the number of valid moves and the 1.9-second time constraint. Near the endgame, the depth increases as fewer moves are available, while in the middle of the game, the depth is limited due to the large number of valid moves. Key components addressing depth include iterative deepening and the timeout mechanism to prevent exceeding the time limit. On larger boards, the agent's depth is more constrained due to the higher number of valid moves.

**Impact on Performance:** Adjusting the search depth within the time constraints allows the agent to adapt its strategy according to different phases of the game.

### 3.2 Search Breadth

**Breadth of Search:** The breadth of the agent's search at any given state is determined by the number of valid moves available after excluding those pruned by the alpha-beta algorithm based on heuristic evaluations. The breadth is similar for both players, as it depends on the number of valid moves for each. For each turn, the agent generates all valid moves for both players and evaluates them.

**Move Ordering and Pruning:** The alpha-beta pruning function leverages move ordering by prioritizing moves based on heuristic evaluations. This ordering increases the likelihood of pruning suboptimal branches early, thereby reducing the effective search space.

### 3.3 Board Size Impact

**Size Considerations:** The agent's approach scales efficiently with the board size. Larger boards naturally increase the number of valid moves, but we mitigate this with move ordering and pruning. As the board size grows, the number of moves the agent can explore increases as well.

**Customizations:**

- **Static Weights:** The `initialize_static_weights` function adjusts weights dynamically based on board size, prioritizing corners and edges while penalizing near-corner positions.
- **Dynamic Weights:** Through the `calculate_dynamic_weights` function, we combine static weights with stability scores, ensuring that the evaluation adapts to both the board size and the current state. This balances short-term gains with long-term stability.

### 3.4 Heuristics, Pruning, and Move Ordering

**Heuristics:** The following heuristics were implemented to evaluate board states and guide the agent's decision-making. Each heuristic was selected based on its relevance to different phases of the game, and their impact was carefully evaluated.

1. **Corner Control Evaluation:** Occupying corners is highly advantageous in Reversi Othello since these tiles cannot be flipped once captured. This heuristic proved to be the most impactful, particularly during the late game, where control over corner positions often determines the outcome. We found that prioritizing corner control dramatically improved the agent's overall performance, especially in critical game phases.
2. **Mobility Evaluation:** By prioritizing higher mobility, the agent seeks to maintain strategic flexibility while limiting the opponent's options. Although this heuristic is useful for assessing the dynamics of the game, it is less impactful compared to the corner control heuristic, especially during the later stages of the game when mobility becomes more limited overall.
3. **Piece Count Evaluation:** It provides a general overview of the game state but has a relatively weaker impact compared to corner control and mobility. The value of this heuristic increases during the endgame, as piece count becomes more critical, directly influencing the score. However, in the opening phase, piece count alone doesn't necessarily reflect control over important squares, making it less significant early on.
4. **Count Capture Evaluation:** This heuristic ensures the agent prefers moves that yield immediate material gain. It is stronger in the later phases of the game when capturing pieces becomes a priority. However, its early-game impact is less pronounced, as it only becomes relevant when there are sufficient valid moves to flip opponent pieces.
5. **Edge Control with Wedging:** This heuristic has a moderate impact but is more effective in the mid- to late-game when edge control becomes more significant.
6. **Stability and Dynamic Weights:** This helps to adjust the evaluation of the board state based on disc stability, rewarding stable positions (such as corners and edges) and penalizing unstable ones (such as those easily flipped by the opponent).

**Pruning and Move Ordering:** We implemented alpha-beta pruning, which reduces the number of nodes evaluated by pruning unimportant branches. This works alongside the `evaluate_board` function to focus on the most promising moves, enhancing search efficiency and ensuring the agent operates within the time limit.

We initially tried ordering moves based on proximity to corners, but this was ineffective early in the game since focusing on corners alone didn't prioritize the most strategically important moves. We switched to ordering moves based on the evaluation function, prioritizing factors like corner control, stability, and mobility, which improved performance.

### 3.5 Win Rate Predictions

**Against a Random Agent:** The agent is expected to win almost every time (98% - 100% win rate during autoplay), as it employs a strategic evaluation of the game state.

**Against Dave (an average human player):** Expected win rate of 70-80%, as the agent adapts to common human mistakes but is not perfect.

**Against Classmates' Agents:** Predictions depend on the strategies of other agents, but the agent is expected to perform within the top 50% due to its strong evaluation heuristics.

## 4. Pros and Cons of Chosen Approach

### 4.1 Advantages

**Alpha-Beta Pruning:** The minimax-based algorithm with alpha-beta pruning improves efficiency by eliminating suboptimal branches and reducing the search space, enabling deeper searches within the time constraints.

**Heuristics:** We use several heuristics, like corner control, mobility, piece count, and edge control, to focus on the most favorable board positions. These heuristics adapt to the game's phase, giving a more detailed and strategic evaluation of each move.

**Fallback Mechanism:** In case of failure to find an optimal move, the agent defaults to a valid move, ensuring it can always play.

### 4.2 Disadvantages

**Time Sensitivity:** While iterative deepening offers flexibility in search depth, it can still lead to suboptimal moves if the time limit is too strict.

**Heuristic Dependence:** The performance of the algorithm is heavily reliant on the heuristics used. In more complex board states, these heuristics may not always lead to optimal decisions. One of the key challenges in developing the heuristics was fine-tuning the weights used in both the heuristics and the game phase evaluation. We couldn't set the weights to their maximum potential because we were unable to run autoplay after each small adjustment. As a result, we had to rely on individual assessments of the game's progress, which, while useful, may not have always been entirely reliable. This made the process of tweaking the weights more time-consuming and less precise, which could have affected the overall effectiveness of the heuristics.

**Simulated Move Sorting:** Sorting simulated moves before evaluation can lead to inefficiencies, especially when there are a large number of valid moves to consider.

### 4.3 Expected Failure Modes

**Shallow Search:** In highly dynamic or complex positions, the agent's search depth might not be enough to fully analyze the board, leading to poor decision-making.

**Timeouts:** If the agent encounters a particularly challenging situation, it might run out of time and be forced to make a less informed move.

**Heuristic Misalignment:** The agent's heuristics may not always capture all relevant aspects of the game, such as the importance of corner control, especially in the middle and endgame where it becomes crucial to secure stable positions. Additionally, the weight tuning may not fully reflect their importance, leading to misalignment in the evaluation.

## 5. Future Improvements

- **Enhanced Heuristics:** There are several ways the current heuristics could be further refined for better performance [1]:
  - **Frontier Discs:** These are discs adjacent to an empty square, and their positions are crucial in determining how many moves a player can make. Minimizing frontier discs and maximizing internal discs—those not adjacent to empty squares—could enhance stability and limit the opponent's options. Implementing this heuristic would involve scanning the board to identify frontier discs and adjusting the evaluation function to prioritize stability by favoring internal discs over frontier ones.
  - **Gaining Tempo:** Tempo refers to forcing the opponent into making moves they would prefer not to, allowing the player to execute critical moves later. By strategically gaining tempo, the player can dictate the flow of the game. This heuristic could be implemented by evaluating positions where forcing the opponent into a suboptimal move could lead to more favorable future options. While harder to quantify, it could be approximated by scoring positions where the opponent is likely to lose control of critical areas of the board.
  - **Stoner Traps:** The Stoner trap is a strategy where the player plays to an X-square to control a diagonal, followed by exploiting a weak edge to force a corner exchange. This can lead to a favorable positional advantage by controlling key areas of the board. Implementing the Stoner trap would involve identifying the right moments to play an X-square, followed by tracking the opponent's edge vulnerabilities and responding accordingly. This could be implemented through pattern recognition or by using strategic evaluation to prioritize moves that lead to forced exchanges.

- **Dynamic Weighting and Stability Refinement:** Dynamic weighting, particularly in the context of stability calculations, has shown potential in this project. It performed better against earlier versions of the program that relied solely on static weights. However, further testing is necessary to confirm its overall effectiveness and to evaluate whether its computational cost is justified. Currently, dynamic weights are computed by multiplying static weights with a stability matrix derived from an analysis of the board. While this provides more nuanced evaluations, it can be computationally expensive due to the need to repeatedly calculate stability for each disk and adjust weights accordingly. To make this approach more efficient, one possible improvement is to cache stability calculations for positions that are unlikely to change during specific phases of the game, such as corners or edges. Additionally, simplifying stability categories or focusing only on areas of the board with higher volatility could reduce the number of computations required without significantly compromising accuracy. These optimizations could make dynamic weighting a more practical tool for real-time gameplay scenarios.

- **Monte Carlo Tree Search (MCTS):** Exploring Monte Carlo Tree Search (MCTS) could be an alternative approach to evaluate more dynamic playstyles, especially in complex or less deterministic situations. MCTS could help the agent make more informed decisions by simulating potential future moves and outcomes in a stochastic manner [2].

- **Adaptive Search Depth:** Adjusting how deeply the algorithm searches could make better use of resources depending on the phase of the game. Early on, when the board is less crowded, a shallower search might be enough since the long-term impact of moves is less clear. Later in the game, a deeper search could provide better insights and ensure smarter decisions. This dynamic adjustment would help balance speed and accuracy based on the current state of the board.

- **Machine Learning Techniques:** Using reinforcement learning could allow the agent to get better over time by learning from its games. This would help it adapt to different strategies and situations, making smarter moves as it gains experience. Adding a caching system to remember and reuse previously evaluated moves could also save time and make decisions faster by avoiding unnecessary recalculations [2].

- **Tuning Heuristic Weights:** One of the challenges identified was the difficulty in determining the optimal weights for the heuristics, given the limited time for exhaustive testing. An improvement would be to automate or semi-automate the process of tuning these weights by running simulations against diverse opponents. This approach could leverage performance metrics to identify the most effective weight combinations, addressing the limitations faced during manual tuning and enhancing the overall evaluation function.

## 6. External Input Source

We started from a prompt given to ChatGPT. The full prompt is shown in Figure 1.

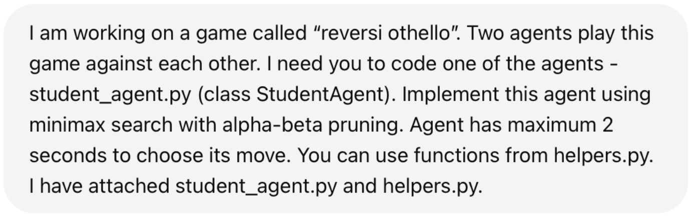
I am working on a game called "reversi othello". Two agents play this game against each other. I need you to code one of the agents - student_agent.py (class StudentAgent). Implement this agent using minimax search with alpha-beta pruning. Agent has maximum 2 seconds to choose its move. You can use functions from helpers.py. I have attached student_agent.py and helpers.py.

Figure 1: ChatGPT Prompt

It gave us a relatively simple code with the `step`, `alpha_beta_search`, `max_value`, `min_value`, and `evaluate_board` functions. Due to their generality, we used most of the given `step`, `alpha_beta_search`, `max_value`, and `min_value` functions with some improvements (added move ordering and made some pieces of code more efficient). However, this reused code only compromised a small portion of the final code. The major part of the final code, including the various heuristics and the evaluation function, was designed and written by us.

In addition to the ChatGPT-generated code, we also referred to a strategy guide for Othello, available at the following link: Strategy Guide. This guide was originally created by Emmanuel Lazard and translated by Colin Springer. It provided us with insights into Othello strategies, which were useful in understanding key game principles such as corner control, edge stability, wedges, and mobility, all of which were incorporated into our heuristics.

## References

[1] Emmanuel Lazard and the French Othello Federation (F.F.O.). Othello strategy. `http://radagast.se/othello/Help/strategy.html`. Accessed: 2024-12-02, Translated from French by Colin Springer, Layout adjusted by Gunnar Andersson, (C) Emmanuel Lazard and the F.F.O., March 1993.

[2] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Series in Artificial Intelligence. Pearson, 4th edition, 2019. Chapter 5 discusses search algorithms like minimax and alpha-beta pruning in detail, including pseudocode and examples for implementing these strategies in two-player games.