Othello: A Strategic Board Game

Othello, also referred to as **Reversi**, is a classic two-player strategy board game played on an **8x8 grid**. The game is renowned for its simplicity in rules but depth in strategy, making it accessible to beginners while offering a challenging experience for seasoned players.

Game Overview

1. **Players and Discs**:
    - The game involves **two players**: one controls the black discs (X), and the other controls the white discs (O).
    - Each disc has two distinct sides: one black and one white. The discs are flipped during the game to indicate a change in control.
2. **Objective**:
    - The primary goal is to have the **majority of the discs on the board showing your color** by the end of the game.
    - Success is achieved through strategic placement of discs to flip the opponent's discs to your color.

Rules of the Game

1. **Initial Setup**:
    - The game begins with **four discs** placed in the center of the board in a **checkerboard pattern**:
    - **Black (X) always moves first**, giving the first player a slight advantage in initial board control.
2. **Placing Discs**:
    - On each turn, a player places a disc of their color on an **empty cell** of the board.
    - The placement must result in at least one of the opponent's discs being **flanked** (sandwiched) between the newly placed disc and another disc of the same color.
    - Flanking can occur:
        - **Horizontally**
        - **Vertically**
        - **Diagonally**
3. **Flipping Discs**:

- Once a valid move is made, all **sandwiched discs** (opponent's discs between two of the player's discs) are **flipped** to the player's color.

4. **Valid Moves**:
   - A move is valid only if it results in at least one opponent's disc being flipped.
   - If a player has no valid moves, they must **pass their turn**.
5. **Game Termination**:
   - The game ends when:
     - Neither player has a valid move.
     - The board is completely filled with discs.
6. **Determining the Winner**:
   - At the end of the game, the player with the **majority of discs showing their color** is declared the winner.
   - If both players have the same number of discs, the game ends in a **tie**.

Game Strategy

Othello's depth lies in its strategic elements. While the rules are simple, achieving victory requires careful planning, foresight, and adaptability.

1. **Controlling Key Positions**:
   - **Corners**:
     - Corners are the most valuable positions on the board because discs placed there are **permanently stable** and cannot be flipped.
     - Controlling corners often leads to dominance in adjacent areas.
   - **Edges**:
     - Edges are moderately valuable as they are more challenging to flip compared to interior positions.
     - Gaining control of the edges can restrict the opponent's mobility.
2. **Minimizing Vulnerability**:
   - Avoid placing discs in positions adjacent to unclaimed corners (e.g., "b2", "g7") early in the game. These positions can enable the opponent to capture the corner, which is highly advantageous.
3. **Maximizing Mobility**:
   - Mobility refers to the number of valid moves available to a player.

- Maintaining high mobility ensures flexibility in gameplay and prevents being forced into unfavorable moves.

4. **Stability vs. Aggression**:
    - While capturing many discs early in the game might seem advantageous, it often leaves the player vulnerable.
    - Focus on **stability** (discs that are unlikely to be flipped) rather than sheer numbers, especially in the early and midgame.
5. **Endgame Control**:
    - As the game nears its conclusion, the number of valid moves diminishes, making each move more critical.
    - Strategic positioning during the midgame can secure dominance in the endgame.

Game Dynamics

1. **Early Game**:
    - Players focus on gaining control of the center of the board.
    - Avoid aggressive expansion that could open access to corners for the opponent.
2. **Midgame**:
    - The game becomes more dynamic as players maneuver to maximize mobility and secure positional advantages.
    - Strategic sacrifices (intentionally giving up discs) may be employed to gain long-term control.
3. **Endgame**:
    - Mobility becomes limited, and players aim to secure stable regions of the board.
    - Tactical planning in the earlier stages directly influences the outcome.

Why Othello is Unique

1. **Dynamic Reversals**:
    - The game often sees dramatic shifts in control, where a player trailing in the midgame can recover and win in the endgame.
2. **Simple Rules, Deep Strategy**:
    - The rules are easy to learn, making the game accessible to beginners.
    - However, mastering the game requires advanced strategic thinking, foresight, and adaptability.

3. **Mathematical and Computational Interest**:
    - Othello is a favorite subject for AI research due to its defined rules, manageable complexity, and requirement for strategic planning.
    - Techniques like **Minimax with Alpha-Beta Pruning** and **heuristics** are widely studied in the context of Othello AI.

Othello is more than just a board game; it is a battle of wits and strategy where each move must be carefully considered. Its combination of simplicity and strategic depth has captivated players worldwide, making it a timeless classic.

Design Document

1. Implementation Details (Classes, Fields, Methods):

The Othello game implementation is structured around several key classes, each responsible for different aspects of the game. Below is a **detailed description** of each class, its fields, and methods.

1.1. Tee Class

- **Purpose**:
    - This utility class enables **simultaneous output** to both the console and a file. It acts as a **write multiplexer** for output streams, ensuring that all output is logged to both destinations in real-time.
    - This is particularly useful for debugging and recording gameplay sessions, as it allows developers to monitor the game's progress while saving a detailed log for later analysis.
- **Fields**:
    - files: A tuple of file objects to which the output is written. This can include the standard output (console) and a file object for logging.
- **Methods**:
    - write(obj):
        - **Purpose**: Writes the given object (e.g., a string) to all registered file streams.
        - **Behavior**: Iterates through the files tuple and writes the object to each stream. After writing, it flushes the stream to ensure immediate output, avoiding buffering delays.
    - flush():
        - **Purpose**: Flushes all file streams to ensure that all data is written immediately.
        - **Behavior**: Iterates through the files tuple and calls the flush() method on each stream.

1.2. Othello Class

- **Purpose**:
    - This is the **main class** that implements the Othello game rules and mechanics. It manages the game board, player turns, move validation, and game state tracking.
    - The class is responsible for maintaining the state of the game, validating moves, applying moves to the board, and determining when the game is over.
- **Fields**:
    - board: A 2D list representing the 8x8 game board. Each cell can contain one of three values: '.' (empty), 'X' (black disc), or 'O' (white disc).
    - current_player: A string indicating the current player ('X' or 'O'). The game starts with 'X' (black) as the first player.
    - move_history: A list storing all moves made during the game. Each move is recorded as a dictionary containing details such as the player, move position, and flipped discs.
    - game_log: A dictionary containing detailed game state and move information. This log is used to record the game's progress, including the initial state, each move, and the final result.
- **Methods**:
    - __init__():
        - **Purpose**: Initializes the game board with the starting position and sets up game tracking.
        - **Behavior**: Creates an 8x8 board with empty cells and places the initial four discs in the center. It also initializes the move_history and game_log fields.
    - get_board_state():
        - **Purpose**: Captures the current board state, including piece positions in algebraic notation.
        - **Behavior**: Returns a dictionary containing the current board configuration and the positions of all discs for each player.
    - algebraic_to_numeric(move):
        - **Purpose**: Converts algebraic notation (e.g., 'e3') to board coordinates (row, column).
        - **Behavior**: Takes a move in algebraic notation and returns the corresponding (row, column) tuple. If the move is invalid, it returns None.
    - numeric_to_algebraic(row, col):
        - **Purpose**: Converts board coordinates to algebraic notation.

        - **Behavior**: Takes a row and column index and returns the corresponding algebraic notation (e.g., 'e3').
    - display_board():
        - **Purpose**: Prints the current board state to the console.
        - **Behavior**: Iterates through the board and prints each row, along with column labels (a-h) for easy reference.
    - is_valid_move(row, col, player):
        - **Purpose**: Checks if a move is valid according to Othello rules.
        - **Behavior**: A move is valid if it is on an empty cell and flanks at least one opponent's disc in any direction (horizontal, vertical, or diagonal).
    - get_valid_moves(player):
        - **Purpose**: Finds all valid moves for the given player.
        - **Behavior**: Iterates through the board and returns a list of (row, column) tuples representing all valid moves for the player.
    - apply_move(row, col, player):
        - **Purpose**: Applies a move to the board, flipping the appropriate pieces.
        - **Behavior**: Places the player's disc on the board and flips all flanked opponent's discs. It also updates the move_history and game_log with details of the move.
    - has_valid_move(player):
        - **Purpose**: Checks if the player has any valid moves available.
        - **Behavior**: Returns True if the player has at least one valid move, otherwise False.
    - switch_player():
        - **Purpose**: Switches the current player to the opponent.
        - **Behavior**: Toggles the current_player field between 'X' and 'O'.
    - count_discs():
        - **Purpose**: Counts the number of discs for each player.
        - **Behavior**: Returns a tuple containing the number of 'X' and 'O' discs on the board.
    - is_game_over():
        - **Purpose**: Checks if the game is over (no valid moves for either player).
        - **Behavior**: Returns True if neither player has any valid moves, otherwise False.

- print_game_status():
  - **Purpose**: Displays the current game status, including scores and the last move.
  - **Behavior**: Prints the current score and details of the last move (if any).
- save_game_log(filename=None):
  - **Purpose**: Saves the complete game log to a JSON file.
  - **Behavior**: Writes the game_log dictionary to a file in JSON format. The log includes the game's start time, end time, all moves, and the final result.

1.3. OthelloAI Class

- **Purpose**:
    - This class implements the **AI player** using the **minimax algorithm** with **alpha-beta pruning**. It supports multiple evaluation heuristics to guide the AI's decision-making process.
    - The AI player can be configured with different search depths and heuristics, allowing for varying levels of difficulty.
- **Fields**:
    - player: The AI's piece color ('X' or 'O').
    - opponent: The opponent's piece color.
    - depth: The search depth for the minimax algorithm. This determines how many moves ahead the AI will look.
    - heuristic: The chosen evaluation function (1-3). This determines how the AI evaluates board positions.
    - nodes_evaluated: The number of nodes evaluated during the search. This is used for performance analysis.
    - start_time: The start time of the move calculation. This is used to measure the time taken to find the best move.
- **Methods**:
    - __init__(player, depth, heuristic):
        - **Purpose**: Initializes the AI player with the specified settings.
        - **Behavior**: Sets the player, opponent, depth, and heuristic fields. It also initializes nodes_evaluated and start_time.
    - evaluate(game):
        - **Purpose**: Evaluates the board position using the selected heuristic.
        - **Behavior**: Calls the appropriate heuristic function (h1, h2, or h3) based on the heuristic field.
    - positional_strategy(game):
        - **Purpose**: Evaluates the position based on weighted board positions.
        - **Behavior**: Uses a predefined weight matrix to assign values to different board positions. Corners are highly valued, while positions next to corners are penalized.
    - mobility_stability(game):

        - **Purpose**: Evaluates the position based on movement options and piece stability.
        - **Behavior**: Combines the number of valid moves (mobility) with the stability of the player's discs (e.g., corner control).
    - count_stable_discs(game):
        - **Purpose**: Counts the number of stable discs (corners and adjacent stable pieces).
        - **Behavior**: Iterates through the board and counts the number of stable discs controlled by the player.
    - minimax(game, depth, alpha, beta, maximizing_player):
        - **Purpose**: Implements the minimax algorithm with alpha-beta pruning.
        - **Behavior**: Recursively evaluates all possible moves up to the specified depth and returns the best evaluation score. Alpha-beta pruning is used to reduce the number of nodes evaluated.
    - find_best_move(game):
        - **Purpose**: Finds the best move for the AI using the minimax search.
        - **Behavior**: Evaluates all valid moves and returns the move with the highest evaluation score. It also prints performance statistics, such as the number of nodes evaluated and the time taken.

---

2. Minimax Algorithm

Analysis of Minimax with Alpha-Beta Pruning in the OthelloAI Implementation

The minimax algorithm with alpha-beta pruning implemented in the provided Othello.py file showcases a structured and efficient approach to decision-making in the game of Othello. Here's an in-depth look at its logic, efficiency, and depth constraints:

---

1. Key Aspects of the Implementation

Minimax Logic

The minimax algorithm seeks to maximize the player's advantage while minimizing the opponent's. It evaluates potential moves recursively, simulating all possible future states up to a certain depth and returning a heuristic evaluation for each. Alpha-beta pruning optimizes this process by eliminating branches that do not need exploration, reducing computational overhead.

Structure

The algorithm operates as follows:

- **Base Case**: If the recursion depth reaches 0 or the game is over, the algorithm evaluates the current game state using the heuristic self.evaluate(game).
- **Valid Moves**: The algorithm retrieves valid moves for the current player using game.get_valid_moves.
- **Pass Turn**: If there are no valid moves, the turn is passed to the opponent without altering the board state.
- **Recursive Exploration**: The algorithm simulates moves and recursively evaluates resulting states:
    - Maximizing Player: Seeks the move with the highest score.
    - Minimizing Player: Seeks the move with the lowest score.
- **Pruning**: Alpha-beta thresholds (alpha and beta) are updated at each step to skip unnecessary branches.

Alpha-Beta Pruning

Alpha-beta pruning reduces the number of nodes evaluated in the minimax tree by skipping branches that cannot influence the final decision. This optimization significantly increases efficiency, particularly in games like Othello with large branching factors.

- **Alpha**: The best score the maximizing player can guarantee.
- **Beta**: The best score the minimizing player can guarantee.
- **Pruning Condition**: When beta <= alpha, further exploration of a branch is unnecessary as it cannot affect the outcome.

---

2. Efficiency Considerations

Reduction in Search Space

Alpha-beta pruning dramatically improves the performance of minimax:

- Without pruning, the algorithm evaluates every possible game state up to the depth limit.
- With pruning, irrelevant branches are skipped, reducing the number of states explored.

The efficiency of pruning depends on the order of move exploration:

- **Optimal Order**: Exploring better moves earlier leads to more pruning.
- **Worst Case**: If poor moves are explored first, pruning is less effective.

Customizable Depth

The depth constraint allows balancing between computation time and decision quality. A higher depth provides better strategic decisions but increases runtime.

Impact of Depth

- **Shallow Depth**: Quick responses but limited foresight.
- **Deep Depth**: Improved strategic planning but higher computational costs.

---

3. Depth Constraint

The recursion depth is controlled by the depth parameter passed to the minimax function. It decreases with each recursive call:

```
eval = self.minimax(new_game, depth - 1, alpha, beta, not maximizing_player)
```

Base Case

The recursion terminates when:

1. The specified depth is 0.
2. The game is over (e.g., no valid moves for either player or the board is full).

At the base case, the heuristic function evaluates the game state:

```
if depth == 0 or game.is_game_over():
    return self.evaluate(game)
```

Dynamic Depth Adjustment

If no valid moves exist, the algorithm reduces the depth and passes the turn:

```
if not valid_moves:
    return self.minimax(game, depth - 1, alpha, beta, not maximizing_player)
```

---

4. Strengths of the Implementation

1. **Efficiency with Alpha-Beta Pruning**
    - By skipping irrelevant branches, the algorithm significantly reduces the number of evaluations, making it feasible to explore deeper trees.
    - In the best case, alpha-beta pruning reduces complexity from $\mathbf{O(b^d)}$ to $\mathbf{O(b^{d/2})}$, where b is the branching factor and d is the depth.
2. **Customizable Depth**
    - The depth parameter allows tuning the algorithm for specific hardware or gameplay scenarios, balancing between decision quality and runtime.
3. **Game-Specific Adaptation**
    - The implementation handles Othello-specific scenarios, such as passing turns when no moves are available, ensuring accurate simulations.

---

3. Evaluation Methods

The evaluation methods (h1, h2, and h3) are heuristic approaches that the AI uses to analyze and score the game board during the Minimax search. These methods guide the AI in determining the optimal move based on specific strategies and priorities.

---

h1 - Basic Disc Count

Formula:

Score = (Player's disc Count) – (Opponent's disc Count)

Explanation:

1. **Purpose**:
    - This heuristic focuses on maximizing the total number of discs for the AI player while minimizing the opponent's discs.
    - It provides a straightforward evaluation that encourages the AI to gain as many discs as possible at any given point in the game.
2. **Strengths**:
    - Simple and efficient: The disc count is easy to calculate and adds minimal computational overhead.
    - Provides a basic metric for evaluating which player is currently in a stronger position.
3. **Weaknesses**:
    - Ignores positional value: It does not consider the importance of strategic positions like corners or edges.
    - Misleading in early/mid-game: In the early stages, capturing many discs might leave the AI vulnerable, especially near corners or edges, where the opponent could capitalize later.
    - Endgame bias: This heuristic is most effective in the late game when the number of discs directly correlates with victory.

---

h2 - Positional Strategy

Weighted Board Values:

The board is divided into zones, each assigned a value based on its strategic importance:

+100 -20 +10 +5 +5 +10 -20 +100

-20 -50 +1 +1 +1 +1 -50 -20

+10 +1 +5 +2 +2 +5 +1 +10

+5 +1 +2 +1 +1 +2 +1 +5

+5 +1 +2 +1 +1 +2 +1 +5

+10 +1 +5 +2 +2 +5 +1 +10

-20 -50 +1 +1 +1 +1 -50 -20

+100 -20 +10 +5 +5 +10 -20 +100

Explanation:

1. **Purpose**:
    - Encourages the AI to control valuable positions, particularly the corners and edges, while avoiding risky areas near corners.
2. **Key Features**:
    - **Corners**:
        - The most stable positions on the board.
        - Once captured, they cannot be flipped, providing long-term stability and control.
    - **Edges**:
        - Moderately valued because they are harder to flip once secured.
    - **Near Corners**:
        - Positions adjacent to corners (e.g., "b2", "g7") are penalized because placing a disc there early can enable the opponent to capture the corner.
3. **Strengths**:
    - Encourages strategic play by focusing on long-term positional advantages.
    - Prevents the AI from making moves that appear strong numerically but are strategically weak.
4. **Weaknesses**:

- Computationally more intensive than h1 due to the need to evaluate positional weights.
- Does not explicitly consider mobility or the number of valid moves.

---

h3 - Combined Mobility and Stability

Formula:

$$\text{Score} = (\text{Mobility} \times 10) + (\text{Stability} \times 30)$$

Components:

1. **Mobility**:
    - Measures the difference in the number of valid moves available to the AI and the opponent.
    - Formula: Mobility=(AI’s valid moves)−(Opponent’s valid moves)\text{Mobility} = (\text{AI's valid moves}) - (\text{Opponent's valid moves})Mobility=(AI’s valid moves)−(Opponent’s valid moves)
    - **Purpose**:
        - Encourages the AI to prioritize moves that maximize its own flexibility while restricting the opponent's options.
        - A higher mobility score indicates a more advantageous position for the AI.
2. **Stability**:
    - Counts the number of stable discs controlled by the AI:
        - **Stable discs** are those that cannot be flipped for the remainder of the game (e.g., corners and discs adjacent to corners with proper support).
    - Stability is evaluated based on:
        - **Corners**: Provide absolute stability and significantly impact the game's outcome.
        - **Adjacent Stable Discs**: Discs supported by stable corners or edges.
    - **Purpose**:
        - Rewards the AI for securing long-term control over critical regions of the board.

Strengths:

- Balances short-term and long-term strategies:
    - Mobility addresses immediate tactical considerations.
    - Stability ensures a strong position for the endgame.
- Discourages risky moves, like placing discs near corners early.
- Favors adaptability and strategic control.

Weaknesses:

- Computational overhead: Requires calculating both mobility and stability, which can be expensive for deeper searches.
- Complexity: Weighing the two factors correctly is crucial; incorrect weights can lead to suboptimal strategies.

Comparison of Heuristics

Heuristic	Focus	Strengths	Weaknesses
h1	Disc count	Simple, fast, and effective in late-game.	Ignores positional importance and mobility.
h2	Positional value	Encourages strategic control of key areas.	Ignores mobility; weaker in dynamic stages.
h3	Mobility + Stability	Balances flexibility and long-term strategy.	Computationally expensive for deep searches.

Each heuristic serves a specific purpose and excels in different stages of the game. For example:

- **h1** is ideal for quick evaluations in the endgame.
- **h2** works well in the midgame, where positional control is critical.
- **h3** provides a balanced evaluation, making it suitable for all stages but at the cost of higher computational demands.

4. Maximum Ply Depth and Time Measurement

The depth of the search tree (referred to as "ply depth") plays a critical role in determining the strength and efficiency of the AI in the game of Othello. The Minimax algorithm with alpha-beta pruning, which underpins the AI's decision-making process, exponentially increases in complexity as the depth increases. This section expands on the concepts of maximum ply depth, associated computational time, and hardware considerations.

Definitions

1. **Depth**:
    - Refers to how far ahead the algorithm looks in the game tree.
    - In Minimax, a depth of n means the AI evaluates all possible moves and countermoves up to n turns into the future.
    - Depth is often set by the programmer or user as a constraint to limit computational effort.
2. **Ply**:
    - A "ply" represents a single move by one player.
    - In **a turn-based game**, a full turn involves one ply from Player 1 and one ply from Player 2 (two plies total).
    - Depth in Minimax is typically measured in terms of plies.

In Minimax:

- Depth = Number of plies to search ahead.
- A depth of 1 means the AI considers only the immediate moves (1 ply).
- A depth of 2 means the AI considers its own move and the opponent's countermove (2 plies).

In Othello Code

- When the depth is set to n, the AI searches n plies (not full turns) into the future. For example:
    - Depth 1: The AI considers its own immediate moves (1 ply).
    - Depth 2: The AI considers its move and the opponent's immediate countermove (2 plies).

Depth in the code is measured in terms of **plies**. For Othello (and other turn-based games), depth is equivalent to the number of plies, not full turns.

Maximum Ply Depth

1. **Definition**:
    - Ply depth refers to the number of layers of moves the AI evaluates in the game tree. A depth of n means the AI considers all possible sequences of n moves (including the opponent's responses) before deciding on its action.
2. **Typical Ply Depths in Practice**:
    - **Shallow Depths (1-4 plies)**:
        - These are suitable for quick decisions but result in suboptimal gameplay since the AI lacks the foresight to evaluate deeper strategic outcomes.
    - **Moderate Depths (5-8 plies)**:
        - Provides a good balance between performance and decision quality. These depths allow the AI to consider multiple scenarios while maintaining reasonable computation times.
    - **Deep Searches (9-12+ plies)**:
        - These depths significantly improve strategic planning, particularly in endgame scenarios. However, they are computationally expensive and may not be practical for real-time gameplay without powerful hardware.
3. **Exponential Growth in Complexity**:
    - Each additional ply effectively doubles the number of potential game states to evaluate, following the formula:

$$\textbf{Total Nodes} = b^d$$

where b is the branching factor (number of valid moves per turn, typically 5-20 in Othello), and d is the ply depth.

    - For example:
        - At depth 5 with an average branching factor of 10, the AI evaluates

$$10^5 = 100{,}000 \text{ nodes.}$$

        - At depth 8, this increases to

$$10^8 = 100{,}000{,}000 \text{ nodes.}$$

requiring significantly more computation.

4. **Maximum Practical Depth**:

- On typical hardware, the AI can handle a maximum depth of **8-10** plies within reasonable time limits for gameplay.
- Beyond this, computation times become excessively long, especially during the midgame when the branching factor is high.

Time Required

1. **Factors Influencing Time**:
   - **Board State**:
     - Early and midgame: Higher branching factor due to numerous valid moves, resulting in longer evaluation times.
     - Endgame: Fewer valid moves reduce the branching factor, making deeper searches feasible.
   - **Ply Depth**:
     - Time increases exponentially with each additional ply, as explained earlier.
   - **Optimization**:
     - Alpha-beta pruning significantly reduces the number of nodes evaluated, especially if the move ordering is efficient.
2. **Typical Time Per Move**:
   - **Depth 6**:
     - Early/Midgame: ~1-2 seconds.
     - Endgame: Less than 1 second.
   - **Depth 8**:
     - Early/Midgame: ~2-5 seconds.
     - Endgame: ~1-2 seconds.
   - **Depth 10**:
     - Early/Midgame: ~10-30 seconds.
     - Endgame: ~5-10 seconds.
3. **Trade-offs**:
   - Higher depths improve decision quality but at the cost of longer computation times. For real-time gameplay, depths of 6-8 are optimal.
   - Precomputing endgame scenarios or using transposition tables can improve efficiency at higher depths.

Hardware Specifications

Used Hardware:

- **CPU**: Intel(R) Core (TM) Ultra 9 185H 2.50 GHz
  - A high-performance processor with multiple cores and threads, suitable for computationally intensive tasks.
- **RAM**: 32 GB
- **Operating System**: Windows 11
- **Python Version**: 3.12.6