Adversarial search

Othello Game

VAHID FARAJI

JANNA OSAMA

February 10, 2025

Contents

1 Introduction

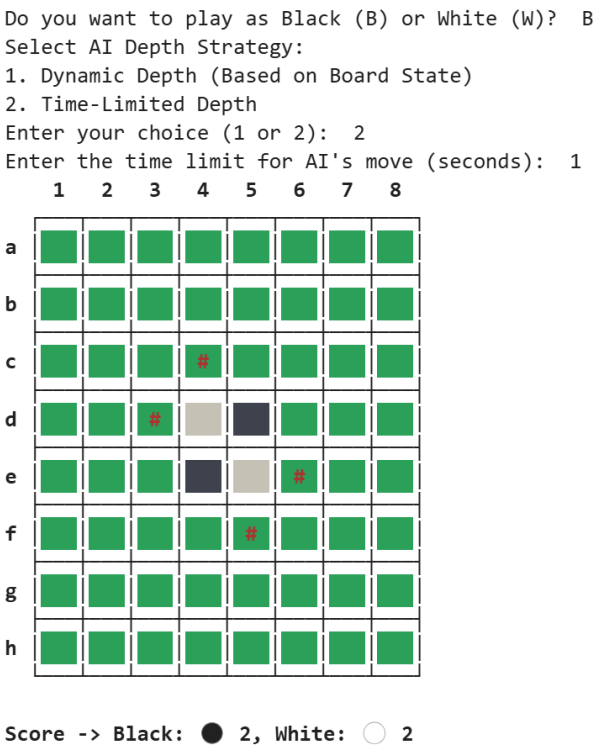
This report details the implementation of an adversarial search algorithm for playing Othello (Reversi) using the Minimax algorithm with Alpha-Beta pruning technique. We developed an AI program that can play a full Othello game against a human opponent until no moves are left or one of the players passes the game, showing the final winner and the scores for both.

2 Implementation Details

The implementation is done in Python using built-in functions like `random` to generate the board.

2.1 Board specifications

8 columns (1 to 8) and 8 rows (a to h). Both players are represented as Black (B) or White (W). There is a green background for the game board which players put their choices on the valid cells.



2.2 Steps on How to Run the Program and Test It

1. Run the python script.
2. The game asks the player to select the colour “B” or “W”.
3. The game asks the user to select an AI strategy (Depth or Time based).
4. The game shows the user the available moves to choose from.

5. The board gets updated with every move, as well as the scores.
6. The game keeps alternating between the AI and the player until no moves are left or one of them passes.
7. Final score is shown with the winner's announcement.

2.3 Core functions

1. `initialize_board()`

Initializes the Othello game board with the standard starting configuration. Places 'W' (White) and 'B' (Black) pieces in the center.

2. `get_square_display(cell, is_valid_move)`

Returns the visual representation of a board square. Displays either a black piece, white piece, an empty green square, or highlights valid moves with a special marker.

3. `print_board(board, valid_moves)`

Prints the current state of the board. Highlights valid moves, updates scores, and formats the board with row and column labels.

4. `get_valid_moves(board, player)`

Generates a list of all valid moves for the current player. Checks for moves that follow Othello's flipping rules.

5. `is_valid_move(board, row, col, player)`

Verifies whether placing a piece at a specific position is a valid move. Checks all directions (horizontal, vertical, diagonal) to see if it leads to flipping opponent pieces.

1. Checks if there are any valid moves and terminates otherwise.
2. Checks if the target position is an empty space.
3. Verifies that moves are either diagonal, horizontal, or vertical.
4. Ensures there is an opponent piece between the original tile and the newly placed one.

6. `execute_move(board, row, col, player)`

Executes a move by placing the player's piece on the board. Flips opponent pieces along all valid directions.

7. `evaluate_board(board, ai_color)`

Evaluates the board state for the AI. Uses predefined weights for board positions (e.g., corners have higher value) to calculate a score.

8. `ai_choose_move(board, ai_color)`

Chooses the best move for the AI using the Minimax algorithm with Alpha-Beta pruning. Adjusts search depth dynamically based on game progress.

9. `get_player_color()`

Prompts the player to choose their color ('B' for Black or 'W' for White).

10. `get_dynamic_depth(board)`

Determines the search depth dynamically based on the number of empty spaces on the board. Uses deeper searches in the late game when fewer moves are available.

11. `get_time_limit()`

Asks the user to input a time limit for the AI's decision-making process. Ensures the input is a positive number.

12. `get_time_limited_depth(board, ai_color, max_depth, time_limit)`

Uses iterative deepening to find the maximum depth the AI can search within a time limit. Returns the deepest fully completed search depth.

13. `select_depth_strategy()`

Allows the player to select the AI depth strategy:

- Dynamic depth based on game state.
- Time-limited depth for faster moves.

14. `format_move_to_xy(row, col)`

Converts board coordinates to a user-friendly format (e.g., `d3`).

15. `parse_xy_move(move_input)`

Converts user input (e.g., `d3`) to board coordinates. Raises an error if the input format is invalid.

16. `minimax(board, depth, alpha, beta, maximizing_player, ai_color, start_time, time_limit)`

1. Recursive algorithm that maximizes the AI player's score and minimizes the opponent's score using Alpha-Beta pruning.
2. Base case: Terminates if no valid moves are available.
3. Explores the game tree until either the time or depth limit is reached, returning the best solution found.
4. Alternates between maximizing (AI) and minimizing (opponent) strategies.

5. Uses Alpha (AI's best guaranteed score) and Beta (opponent's best guaranteed score) to eliminate unnecessary branches, improving efficiency.

6. If $\beta \leq \alpha$, further exploration is skipped.

17. `game_loop()`

The main function that runs the game. Initializes the board and handles the player vs. AI game flow. Alternates turns between the player and AI. Ends the game when both players pass consecutively or there are no valid moves left.

2.4 Main Game Loop

1. The game alternates turns between the player (showing the move format, e.g., `f3`) and the AI.

2. If a player has no valid moves, they must pass. If both players pass consecutively, the game ends.

3. The board is displayed at the start of each turn.

2.5 Background on Minimax and Its Efficiency in Othello

Minimax is a type of backtracking algorithm that is used in designing AI models for games. It is efficient because it assumes that both players play optimally. The players are a maximizer that tries to get the highest score while the minimizer tries to get the lowest score possible. The AI agent (assuming it is the maximizer) explores all possible movements and then backtracks, choosing a decision based on successful movements. If the maximizer has a successful move, it adds a positive number to the score. Conversely, if the minimizer succeeds, a negative number is added to the score.

2.6 Implementing Alpha-Beta Pruning with Minimax Algorithm

Alpha-Beta pruning enhances the efficiency of the Minimax algorithm by reducing its time complexity. It allows the AI agent to focus only on optimal moves, cutting off other branches and saving the best decision until a better one is found. This accelerates the search process by using Alpha (the best maximizer value) and Beta (the best minimizer value). The key pruning condition is:

$$\text{if } \beta \leq \alpha$$

2.7 Key Features in the Program

2.7.1 Position Weights and Their Role in Evaluation

Position-based heuristic prioritizes board positions such as corners, which have higher values reflecting stability. The `POSITION_WEIGHTS` matrix defines weighted values for each position on the Othello board. These weights reflect the strategic importance of certain positions, guiding the AI to make better decisions. The primary goals of using this matrix include:

1. **Prioritizing Stable Positions:** Corner positions, which cannot be flipped once captured, are assigned the highest values (e.g., 100). These positions provide long-term stability in the game.

```
POSITION_WEIGHTS = [
    [100, -20, 30, 10, 10, 30, -20, 100],
    [-20, -30, -2, -2, -2, -2, -30, -20],
    [ 30,  -2, -1, -1, -1, -1,  -2,  30],
    [ 10,  -2, -1,  0,  0, -1,  -2,  10],
    [ 10,  -2, -1,  0,  0, -1,  -2,  10],
    [ 30,  -2, -1, -1, -1, -1,  -2,  30],
    [-20, -30, -2, -2, -2, -2, -30, -20],
    [100, -20, 30, 10, 10, 30, -20, 100]
]
```

Figure 1: position Weights matrix

2. **Discouraging Risky Moves:** Positions adjacent to corners or edges, which can easily be captured by the opponent, are given negative values (e.g., -20 or -30). This discourages the AI from making moves that could jeopardize its stability.

3. **Encouraging Controlled Expansion:** Mid-range positions have moderate values, guiding the AI to develop board control without taking unnecessary risks early on.

During board evaluation, the AI calculates a score based on the sum of weights for positions occupied by its pieces, subtracting the opponent's score. This heuristic helps the AI prioritize moves that maximize its strategic advantage over the long term.

2.7.2 Dynamic Depth Adjustment

The `get_dynamic_depth(board)` function determines the search depth for the Minimax algorithm based on the current game state. Its purpose is to balance performance and decision-making efficiency by adjusting the search depth according to the number of empty spaces on the board. The function works as follows:

1. **Early Game (More than 50 empty spaces):** The search depth is set to 3 to ensure quick decisions, as there are many possible moves, and deep searches are computationally expensive.

2. **Mid Game (Between 30 and 50 empty spaces):** The search depth increases to 5, allowing the AI to make more informed decisions as the board starts to fill up.

3. **Late Game (Fewer than 30 empty spaces):** The search depth is set to 7, enabling the AI to thoroughly evaluate endgame scenarios where precise moves are critical.

This dynamic depth adjustment improves the AI's performance by reducing computation time early in the game and allowing deeper, more strategic searches in later stages.

2.7.3 Time-Limited Depth Search

The `get_time_limited_depth(board, ai_color, max_depth, time_limit)` function dynamically adjusts the search depth for the Minimax algorithm based on a given time limit. Its purpose is to optimize the AI's decision-making within a time-constrained environment using iterative deepening. The function operates as follows:

1. **Iterative Deepening:** The function starts with a shallow depth (1) and progressively increases it up to a maximum depth. This allows the AI to refine its decision-making as long as time permits.

2. **Time Limit Enforcement:** If the time spent exceeds the given limit, the function halts further deepening and returns the last successful search depth.

3. **Fallback Strategy:** If a timeout occurs, the AI uses the best move found at the last successful depth, ensuring that decisions are always made within the available time frame.

This approach provides a balance between thorough decision-making and time efficiency, enabling the AI to handle scenarios where computation time is limited.

2.7.4 Move Selection and Board Visualization

The game board provides a graphical interface to assist the player in selecting valid moves. After each turn, the board is updated with the following key elements:

1. **Valid Move Suggestions:** Squares where the player can make valid moves are highlighted with a bold red '#' symbol. These suggestions are dynamically generated based on the current game state.

2. **Player Interaction:** The player is prompted to select a valid move from the displayed suggestions. The valid moves are also stored in a set and checked to ensure the player's choice is valid.

3. **Board Updates:** Once the player selects a move, the board is updated, and pieces are flipped according to the game's rules. The AI or the opponent then takes their turn.

4. **Score Display:** The current score is displayed at the bottom of the board, showing the number of black and white pieces on the board.

References

[1] GeeksforGeeks. (2023, January 16). *Minimax Algorithm in Game Theory — Set 4 (AlphaBeta Pruning).* Retrieved from:`https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/`

[2] GeeksforGeeks. (2022, June 13). *Minimax Algorithm in Game Theory — Set 1 (Introduction).* Retrieved from: `https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/`

[3] Simon Kristoffersson's Adversarial Search Lecture Notes (2025).

[4] Using ChatGPT in order to address some programming issues, which I got used to searching on `Stackoverflow` website.

[5] Using ChatGPT in order to address some Latex issues, and helping write the report more tidy.