**Othello**

Prof. Snehasis Mukherjee

**Introduction:**

Othello is a strategy board game that was invented in 1883 by a British man named Lewis Waterman, who initially called it "Reversi", later named Othello after the famous Shakespearean play. It's played by two players who take turns placing their colored discs on the board. The game starts with four discs placed in the center in a specific pattern. Each player then alternates placing one disc at a time on the board. When you place a disc, you must "sandwich" your opponent's discs between your newly placed disc and your existing discs. When you sandwich your opponent's discs, you flip them to your color.

The game continues until the board is full or no more moves can be made. The player with the most discs showing their color wins.

**Problem analysis**

The problem analysis involves managing the game state, ensuring valid moves are made, and determining the winner. The game loop simulates the back-and-forth play between the two players until a terminal state is reached. The agents (agent\_X and agent\_O) are responsible for deciding the next move for their respective players based on the current board state. This algorithm holds the concept of the Othello game, where players aim to maximize their tile count on the board and strategically flip opponent tiles to gain an advantage. The use of agents and the game loop structure allows for a systematic and organized approach to playing the game.

**Solver Ideation**

The process of ideating an Othello solver incorporates the development of AI agents aimed towards strategic gameplay. First, it focuses on an evaluation function that analyzes the game state, factoring in elements like piece count, positional advantages, and potential future moves. Adding the Minimax algorithm, a recursive search algorithm tailored for two-player games administrates decision-making. The integration of Alpha-Beta pruning optimizes the Minimax algorithm, optimizing the search process. Refinement of heuristics, with a focus on Othello-specific characteristics such as stability arrays, further enhances the AI. Some testing and iteration against opponents, including different AI difficulties and players, refine the solver's performance. Finally, integration into a user interface (UI) ensures a proper depiction of our game.

Data structures used

-**2D Array**: A 2D array is commonly used to represent the Othello game board.

- **Tuple or Pair**: Moves are often represented as tuples or pairs of integers, indicating the row and column where a player makes a move on the board.

- **Minimax Tree:** The game tree is often represented as a tree data structure, where each node corresponds to a game state, and edges represent possible moves.

- **Heuristic Weights**: Constants or data structures that store the weights assigned to different factors in the evaluation function.

- **List or Array:** To store a list of possible moves generated during the search process.

- **Hash Table**: A hash table can be used to implement a transposition table for caching previously computed evaluations of game states.

**Deadlock Conditions used:**

A deadlock in the context of a game solver, such as an Othello solver, typically refers to a situation where the search algorithm cannot proceed because it's stuck in a loop or cycle. In the context of game tree search algorithms like Minimax, Alpha-Beta Pruning, or other heuristic search algorithms, deadlocks are not explicitly used. However, certain conditions may lead to situations that can be considered deadlock-like. Here are some conditions that might resemble deadlock in a game-solving context:

**1. Repetition of Game States**

- If the search algorithm encounters a repetition of game states during its exploration, and there is no mechanism to handle repeated states, it can lead to a cycle where the same positions are being evaluated repeatedly.

1. **Infinite Loop in Evaluation**

- If the evaluation function or heuristic used in the search process contains a flaw, it might result in an infinite loop, preventing the algorithm from progressing further.

**3. Lack of Terminal Conditions:**

- If the search algorithm does not have proper terminal conditions to end the search (e.g., maximum depth reached, a winning state detected), it may continue exploring indefinitely, which can be considered a form of deadlock.

**ALGORITHM**

Othello can be an interesting game when explored algorithm-wise. After going through a wide range of algorithms we confined ourselves to the minimax algorithm as it fulfilled the needs of the game perfectly and molded perfectly into our plans of action.

Now, the question that arises is how exactly this algorithm helps our case. The minimax algorithm is a recursive algorithm used for decision-making in two-player turn-based games. The work of this algo is to explore all the possible states in the game tree, keep assigning the respective scores to each move and then based on those results, returns the move that has the highest score and is hence the most optimal move for that particular turn. The goal at every move is to maximise the score of the player whose turn it was and also keeping into consideration that the opponent also plays optimally.

The algorithm continues to explore deeper levels of the game tree until a certain depth or a terminal state is reached. Scoring is performed at the leaf nodes, and the scores are propagated up the tree.

**Explanation**:

- If the depth is 0 or the game is over, the `evaluate` function is called to get a score for the current board position.

- If the current player is the maximizing player, it tries to maximize the score by exploring its possible moves recursively.

- If the current player is the minimizing player, it tries to minimize the score by exploring the opponent's possible moves recursively.

- Alpha-beta pruning is used to eliminate branches of the game tree that won't affect the final decision.

Scoring Metrics:

Using an algorithm is not enough and hence we need to provide a set of metrices based on which the algorithm can score the move. Now this scoring metric is made specifically for Othello keeping in mind what move and what location is considered to be the strongest in terms of the game. The game works by scoring every move either positively or negatively. The one having the maximum score wins at the end. Hence, more positive score is considered a boon for this game.

Some major functions to grade are:

- \*Interior Disks: Positive score for maximizing the number of interior disks (disks surrounded by other disks).

- \*Frontier Disks:\* Negative score for minimizing the number of frontier disks (disks with at least one empty tile around them).

- \*Stable Disks:\* Positive score for maximizing the number of stable disks (disks that can't be flipped back).

- \*Tile Positioning:\* Utilizes the `stability` array to assign scores based on the importance of different positions on the board.

- \*Victory:\* A very high positive score if a move leads to victory.

Now consider the matrix below. This matrix tells exactly how a move is scored. Every tile has a different score as you can see in the matrix below. One thing to observe is that we divide the grd into 4 quarters, each of which has the same scoring scheme. Now look closely, and you’ll find theat the corners have the ost positive score. This is because once you get the corner, the opponent cannot take it and hence it effectively becomes a “stable disk”.

Now, every tile surrounding the corner is hugely negative, this is because taking this tile can prove to be really good for your opponent meaning this is the absolute worst choice that a player could’ve made and hence results in the player losing their marks.

The next thing to take into consideration are the edge cases (scores- 10 & 5) are called edge conditions and are considered to be a good move.

Lastly, the center tiles are considered to be mildly bad as it gives the opponent control of the game and hence slightly negative values are used to define them.

stability = [

[100, -20, 10, 5, 5, 10, -20, 100],

[-20, -50, -2, -2, -2, -2, -50, -20],

[10, -2, -1, -1, -1, -1, -2, 10],

[5, -2, -1, -1, -1, -1, -2, 5],

[5, -2, -1, -1, -1, -1, -2, 5],

[10, -2, -1, -1, -1, -1, -2, 10],

[-20, -50, -2, -2, -2, -2, -50, -20],

[100, -20, 10, 5, 5, 10, -20, 100]

]

Negative Score for Opponent's Move:

1. **Minimizing the Opponent's Interior Disks:** The opponent's move may lead to an increase in the number of interior disks (disks surrounded by other disks), which is undesirable for the current player.
2. **Maximizing the Opponent's Frontier Disks:** The opponent's move might increase the number of frontier disks (disks with at least one empty tile around them), providing the opponent with more strategic options.
3. **Minimizing the Opponent's Stable Disks:** If the opponent's move leads to stable disks (disks that can't be flipped back), it could result in a disadvantage for the current player.
4. **Preventing Victory for the Opponent:** If the opponent's move contributes to a winning position, the negative score reflects the urgency of blocking or countering such moves.

**Balancing Positive and Negative Scores:**

The combination of positive scores for maximizing the player's advantageous positions and negative scores for minimizing the opponent's advantageous positions ensures a strategic evaluation of each move. The algorithm aims to find a balance between advancing its own position and hindering the opponent's progress.

**Code flow**

1. **Initialising the file (main.py):**

The program first initializes the Othello board (`OthelloBoard`), and then the two players, player 'X' (`OthelloAgent`), and player 'O' (`OthelloAgent`).

1. Game loop:

Now, we move on to he loop of the game continues until the game is over. Now this is only possible when both the players do not have a valid move and hence the game cannot proceed further.

**Player X turn:**

* The `get\_move` method of the 'X' player agent is called to determine the next move.
* Now the first thing that is checked is the availability of valid moves. If present the chosen move is simply applied to the board using the `make\_move` method.
* But in case any valid move is not present then the turn for that player is skipped and the game progresses normally from there on.

After every move, the board is updated with the tiles in their respective position. This is where the GUI comes into play.

**Check for Game Over:**

After every move the program checks if the game is over. If it does, the loop breaks and we have our result. If not the turn shifts to the other player.

Now same steps are repeated for the other player and the game progresses thereon.

1. **Determine Winner and Display Result (`main.py`):**

* So the rules of the game are easy, the player having the maximum number of tiles on board after the game has ended, wins.
* If both players run out of legal moves then the game ends and the number of tiles are counted right after that to decide the winner.
* The result is then displayed and the suspense is broken.

1. **Player Strategies (`agent.py`):**

* The `OthelloAgent` class defines the strategy for each player using the Minimax algorithm.
* The `get\_move` method is responsible for determining the best move for a player.
* The Minimax algorithm with alpha-beta pruning is used to explore all the possible moves and find the one that is the most optimal.

1. **Scoring Mechanism (`agent.py`):**

* As explained in the algorithm section, the code uses some scoring metrics. These scoring metrics take into account the factors such as maximizing interior disks, minimizing frontier disks, maximizing stable disks, acquiring corners, which are made specifically for Othello. This makes the result more accurate.
* The `evaluate` function assigns a score to a given board state for a specific player.
* The `get\_tile\_score` function assigns scores based on the position of a tile on the board as every tile has a different score.

1. **Minimax Algorithm (`agent.py`):**

* The `minimax` function implements the Minimax algorithm with alpha-beta pruning which is explained in detail in the algorithm section.
* To outline what the algorithm does is that it explores all possible moves up to a specified depth, evaluating each move using the `evaluate` function.
* The algorithm determines the best move for the current player by considering the scores of possible moves.

1. **Game Rules (`game\_rules.py`):**
   * The `get\_possible\_moves` function determines the legal moves for a player based on the current board state.
   * The `make\_move` function updates the board based on a player's move, flipping disks according to Othello rules.

By following this code flow, the program is able to simulate a game of Othello between 2 players both of which are controlled by the AI agent developed by us. The model is trained by using Minimax algorithm and a sophisticated scoring mechanism. The game continues until there are no legal moves for both players or until the board is full.

**Optimization of memory use**:

Optimizing memory usage is crucial in developing efficient algorithms, especially in the context of game-solving algorithms like Minimax for Othello. Here are some strategies to optimize memory use:

**Transposition Tables:**

- Use transposition tables to store previously computed positions along with their evaluations. This reduces redundant computations by storing the results of previously visited states and reusing them when the same positions are encountered again.

**Iterative Deepening:**

- Implement iterative deepening to gradually increase the search depth. This allows you to reuse the results of shallower searches at deeper levels, reducing the need to store evaluations for all nodes at the maximum depth.

**Dynamic Memory Allocation:**

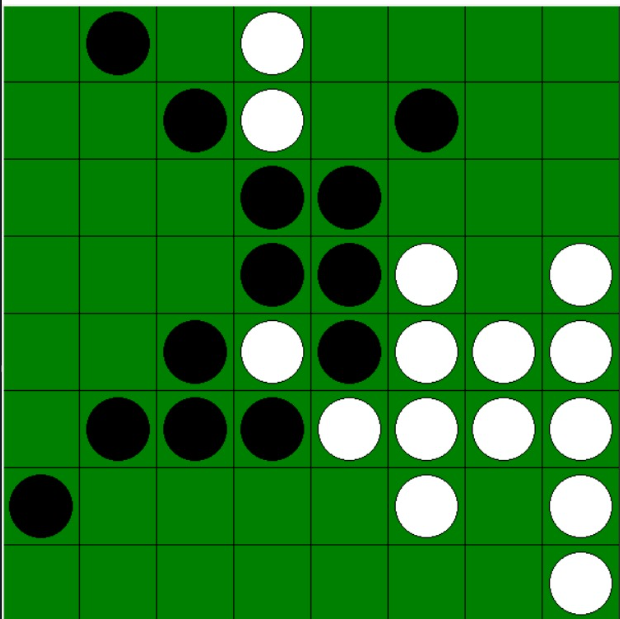
- Allocate memory dynamically for data structures that vary in size during the execution. For example, use dynamic arrays or linked structures instead of fixed-size arrays when the size is not known in advance.

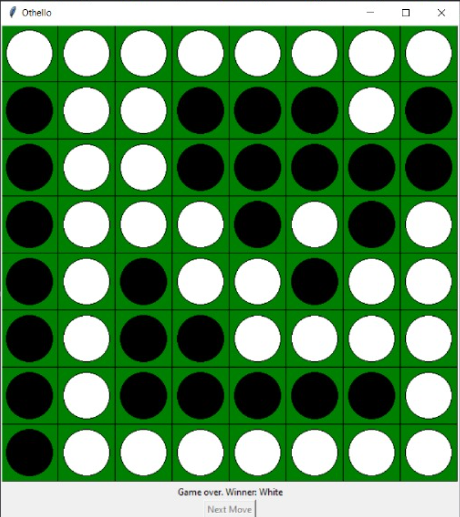
**Our code on test levels:**

Following are the screen snips from our working code which tells you exactly how the look and feel of your code is.

A screenshot of a game

Description automatically generated





References

* [Minimax - Wikipedia](https://en.wikipedia.org/wiki/Minimax)
* [Alpha–beta pruning - Wikipedia](https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning)
* [Transposition Table - Chessprogramming wiki](https://www.chessprogramming.org/Transposition_Table)
* [Game tree - Wikipedia](https://en.wikipedia.org/wiki/Game_tree)
* [pygame news](https://www.pygame.org/news)