Othello Implementation

AI CSD 311

Group 5

2024

Contents

Introduction

Othello is a board game(a very addictive one, as you shall see after playing a few times) that has been derived from Go. It is a two player game, and is played on an 8x8 board. It was patented in Japan in 1971 by Goro Hasegawa, a Japanese salesman. The name Othello is a reference to a Shakespere play, Othello, because the pieces on the board were thought to resemble the characters Moor Othello(who is black) and Desdemona(who is white) and their conflict. This project aims to implement some of the algorithms that were taught in the AI course(and some that weren't, no one stopped us) at SNIoE to play Othello.

Rules of the Game

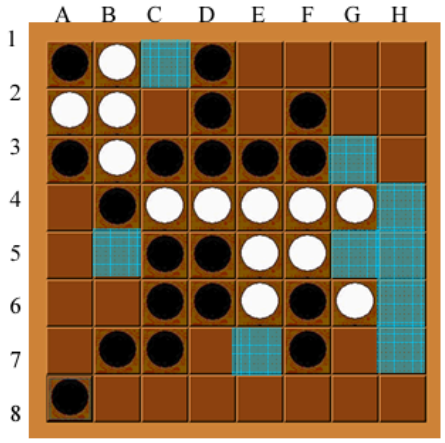
Game Setup

The board starts with four pieces placed in the centre: two black and two white arranged diagonally.One player uses black pieces, and the other uses white. According to the traditional gameplay, black always moves first.

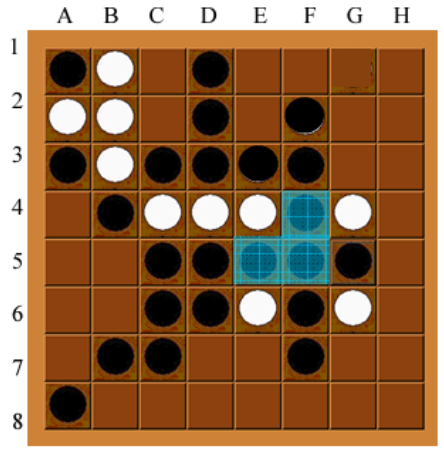
Gameplay

Placing Pieces:

Players take turns placing their piece on an empty square.The placed piece must capture opponent pieces by enclosing them between the newly placed piece and another piece of the same colour, in a straight line (horizontally, vertically, or diagonally).



Blue highlighted squares show valid moves for black.



Blue highlighted squares show the flanked coins once the black has been placed in (G, 5)

Flipping Pieces:

All opponent pieces caught between the two enclosing pieces are flipped to the current player's colour.Multiple lines of pieces can be flipped in one move if conditions are met in different directions.

Valid Moves:

A move is valid only if it results in flipping at least one opponent piece.If no valid moves are available, the player skips their turn.

End of Game:

The game ends when neither player can make a valid move, typically when the board is full or all remaining squares cannot result in flips.

Winning the Game

Once the game ends, count the pieces on the board.The player with the most pieces of their colour wins. If both players have an equal number, the game is a tie.

Strategies

Control the **corners**, as pieces in these positions cannot be flipped.Avoid moves that give your opponent access to advantageous positions.Plan moves to maximise flips while minimising opportunities for your opponent.

These strategies are what we will plan to convert into heuristics for our algorithms.

Translating it to Code

The implementation was pretty straightforward. We made an 8x8 array for the board, labelled the 'open' positions as 0, black coins were labelled as 1, and white coins were labelled as -1. This approach helped count the coins and detect who the players were as the opponent could be defined as -(player). While displaying the board, we used emojis to replicate the actual board. 1 was replaced by a black circle, -1 by a white circle, and 0 by a green square.

We defined a class called 'Othello', and inside it, we defined the various functions we would need. Some key functions were the available_moves function and the valid_moves function. The available_moves function gives the positions on the board not occupied by any player, and it is important to note that all the available moves are not the same as the valid moves. A move is only considered valid when you can move to the position whilst taking over an opponent's coin. After the player selects a move, from the valid moves, the move is made through the make_move function, which positions the coin on the board while also flipping the opponent coins. The turns would keep switching as long as the game is not over.

The Algorithms were also defined in the same class. We implemented Greedy Search with 2 different heuristics, Simulated annealing, and the Minimax algorithm(which turned out to be a dud). This gives the user the ability to play against different levels of difficulty.

Not implementing this code in GUI turned out to be helpful from a coding point of view as we could track if any mistakes where being made in each move, or if the available moves were being displayed correctly. Putting it in GUI would have not given us access to see previous moves or track the game as it progressed.

Heuristics

The choice of heuristics plays a very important role and can help improve the efficiency of the algorithm.

We first implemented the most obvious- and much later to our dismay-the most amateur heuristic to consider according to many papers and Othello enthusiasts, and the strategy that most first time players of Othello employ-observing the number of coins flipped for a particular move.

We struck gold when we found a paper that discussed various algorithms and heuristics that could be employed to better computer moves, and it gave to us a much better heuristic, the weighted board heuristic. This heuristic assigns weights to all the positions on the board based on the advantage that the location offers(e.g. corner locations have the highest values as coins that occupy them cannot be captured). This turned out to be a very practical and simple heuristic, but as it turned out, it did not perform better than the coin flips heuristic, at least when applied to Greedy search, and we shall see more on that later.

H1 - Coin Flips:

- Focus: Maximizing the number of opponent coins flipped in the current turn.
- Relevance:
    - It creates an immediate advantage by increasing the player's control over the board.
    - Easy to compute, as it only involves local evaluation of a move.
- This was updated later on to include Corner Capture, which means that the algorithm prioritises the number of coins flipped unless it can place a coin in any of the corners. In that case, the corner-move is prioritised over all other options.

H2 - Weighted Board:

- Focus: Prioritizing moves based on positional strength using the weight matrix:
    - High-Value Positions:
        - Corners: High weight as they are stable and cannot be flipped.
        - Edges: Moderately high weight as they are harder to flip.
    - Low-Value Positions:
        - Inner tiles, particularly near corners, are penalized since they risk giving the opponent access to stable positions.
- Relevance:
    - Adds strategic depth by encouraging moves that are less likely to leave the player vulnerable.
    - The weight matrix incorporates board dynamics, balancing immediate gains and positional advantage.

Search Algorithms

Greedy Search

Greedy Search is a strategy that selects the move providing the most immediate gain, focusing on maximizing the value of the current step without considering the broader consequences. This approach does not look ahead to future turns but evaluates moves based on a predefined heuristic to determine their desirability.

We have implemented Greedy Search with two heuristics:

1. Greedy Search with H1 (Coin Flips Heuristic)
2. Greedy Search with H2 (Weighted Board Heuristic)

Greedy Search with H1 (Coin Flips Heuristic)

- This evaluates moves based on the number of opponent coins flipped during the current turn.
- We also added the **extra condition of preferring the corner weights**, making it slightly better than the average amateur, but yeah, it could be argued that within a few games against the algorithm , the approach can easily be detected.
- Implementation:
    - The function greedy_search_H1 iterates through the available moves.
    - For each move, it calculates the number of opponent coins flipped using count_flips.
    - The move that flips the maximum number of opponent coins is selected.

Greedy Search with H2 (Weighted Board Heuristic)

- This evaluates moves based on their position on the board, prioritising moves on higher-weighted tiles.
- Implementation:
    - A weight matrix defines the importance of each tile on the board (corners have the highest weight, edges are less valuable, and interior positions are even less valuable).
    - The function greedy_search_H2 iterates through the available moves.
    - For each move, it checks the tile's weight in the weights array.
    - The move with the highest weight is selected.

Both heuristics provide straightforward decision-making mechanisms, making them computationally efficient.

H1 is a very amateur-like heuristic, as the move which gives the most number of coins may not always be the best move, and it may just be your opponent sacrificing some coins to land you in a vulnerable position.

H2 adds a layer of positional awareness, especially useful in mid to late-game scenarios where stability matters.

When we played Greedy H1 against Greedy H2, Greedy H1 won the game, which was not expected since it is an 'amateur' heuristic, but also since we added an extra condition to greedy H1, it wasn't as bad(we gave preference to the corners). This tells us that in the start to mid levels of the game, Greedy H1 is actually a better heuristic than the weighted positions H2. The H2 heuristic is more of a defensive strategy, and thus it is more prone to losing out in the initial stages against H1, as it is difficult to protect fewer coins.

The disadvantages that greedy search offered was that within a few games, the approach used by the algorithms can be detected(may not be as easy in H2), and that can be used against the algorithms.

Simulated Annealing

Simulated annealing is an algorithm that is most known for it's ability to counter the fallibility of most optimization algorithms of getting stuck at local optima points.

How it works:

1. Initial Solution:
    - The algorithm starts with an initial solution (can be random or based on heuristics).
2. Objective Function:
    - Evaluates the quality (or cost) of a solution.
3. Neighbour Solutions:
    - At each step, the algorithm explores neighbouring solutions (small modifications to the current solution).
4. Temperature:
    - Controls the likelihood of accepting worse solutions during the search.
    - Starts high and gradually decreases (cooling schedule).
5. Acceptance Probability:
    - Even if a neighbouring solution is worse (higher cost), it might be accepted with a certain probability:

$$P = e^{-\Delta E/T}$$

- ΔE: Difference in cost between the current and neighbouring solutions.
- T: Current temperature.

This helps escape local minima by allowing the algorithm to explore diverse solutions.

Simulated Annealing in AI

Simulated annealing in the neural network does not directly solve for the "best" strategy in a single instance but instead:

1. Trains the model to **generalise optimal decision-making** for various board states.
2. Balances between exploration (testing suboptimal or less-explored moves) and exploitation (optimising based on current knowledge).

During training, SA helps the AI explore the game tree effectively by occasionally selecting suboptimal moves to avoid getting stuck in local optima.

1. **Exploration (Random Moves)**:
   - At higher "temperatures" (early training epochs), the AI selects moves more randomly.
   - This ensures the model explores a diverse range of board states, which is crucial for robust training.
2. **Exploitation (Greedy Moves)**:
   - As the "temperature" decreases (later epochs), the AI increasingly chooses moves that maximise immediate rewards based on the model's predictions.
   - This helps refine the AI's understanding and focuses on optimal play.
3. **Cooling Schedule**:
   - The exploration rate gradually decreases over training epochs, following an exponential decay: exploration rate=$\max(0.1, 0.99^{\text{epoch}//500})$
   - Early on, 99% exploration ensures the AI learns broadly. Later, 10% randomness allows for fine-tuning while still occasionally exploring alternatives.

While simulated annealing only used a heuristic to flip the most coins per turn, it was still able to make a broader range of decisions due to the uncertainty inherently present in the algorithm. When paired with the H1 heuristic (most coins flips plus corner preference) it outperformed Greedy H1 and H2. The neural network trained for 10.,000 instances of games and outperformed all three algorithms. The heuristic here was not needed as backpropagation was used.

Minimax

The minimax algorithm is a decision-making tool used in adversarial games (like Othello) to determine the optimal move for a player. It assumes that both players are playing optimally and aims to maximise the current player's advantage while minimizing the opponent's advantage.

How Minimax Works

1. Tree Exploration:
    - The algorithm constructs a game tree where nodes represent game states and edges represent possible moves.
2. Maximising and Minimising:
    - The algorithm alternates between *maximising* the current player's score and *minimising* the opponent's score.
3. Evaluation Function:
    - At a certain depth (or at terminal states), the algorithm evaluates game states using a heuristic function. This function assigns scores to game states based on the player's advantage.

Using Minimax in Othello

1. Move Simulation:
    - For each possible move, the algorithm simulates placing a piece, flipping opponent pieces, and updates the board state.
2. Heuristic Function:
    - The evaluation function estimates the desirability of a board state using features like:
        - Piece Count Difference: The difference between the number of black and white pieces.
        - Corner Control: Prioritising corner tiles as they are stable and can't be flipped easily.
        - Mobility: Maximising the number of legal moves available while minimising the opponent's options.
        - Edge Stability: Favouring stable edge pieces that can't be flipped easily.
3. Depth Limit:

- Due to the exponential growth of the game tree, a depth limit is set to stop the recursion and use the evaluation function instead of fully traversing to terminal states.

Effectiveness Of the Algorithms

1. Greedy Search:
    - Greedy Search looks only one move ahead and chooses the move with the immediate maximum gain (flipped pieces).
    - Minimax considers multiple layers of moves, leading to more strategic decision-making by anticipating the opponent's response.
2. Simulated Annealing:
    - This is a very simple and effective because it also intoduces an element of randomness, making it dificult to predict thhe moves even after several gameplays.
3. Alpha-Beta Pruning (an optimization of Minimax):
    - Alpha-Beta pruning reduces the number of nodes evaluated in the tree by eliminating branches that cannot influence the final decision. This makes Minimax faster and more practical for deeper searches.

Minimax with appropriate heuristics significantly outperforms naive strategies (like greedy search or random moves) because Othello is a game where long-term planning is critical.

However, the effectiveness depends on:

- Depth of Search: A deeper search yields better results but requires more computational power.
- Quality of Heuristics: Good heuristics make the algorithm efficient and closer to optimal decisions.
- Game Complexity: In late-game scenarios, the number of legal moves and board configurations reduces, making Minimax more accurate.

Much to our surprise, greedy H1 won against every other algorithm when we played them against each other. This allows us to conclude that, the heuristic is not as amateur as it seems, and since it is the easiest to implement, it is useful to play against, however, there is no randomness, and the moves can easily be predicted after a few gameplays.

Algorithm	Matches Played	Wins	Losses	Ties	Points
H1 (Greedy Search with Corner Strategy)	3	3	0	0	3
H2 (Greedy Search with Positional Heuristics)	3	1	1	1	1.5
Simulated Annealing	3	1	2	0	1
Minimax	3	0	2	1	0.5

ML and Deep Learning Implementation

Artificial Neural Network

Neural Networks are characterised by the layers that define the function it uses for optimisations, These are optimised parametrically using the data fed into the model. THe input layer, for example,represents the current state of the Othello board. In our case, an 8x8 grid is flattened into a vector of 64 inputs, where each input corresponds to a square on the board:

- `1`: Player's piece.
- `-1`: Opponent's piece.
- `0`: Empty square.

Layers between the input and output consist of neurons performing weighted computations followed by non-linear activation functions (e.g., ReLU). These layers extract features and patterns, such as advantageous positions or potential moves. The output layer produces a value for each square on the board, representing the predicted reward (or utility) for placing a piece on that square.

This model contains:

- 3 fully connected layers:
    - Input layer takes the flattened board state (8x8 ↠ 64 features).
    - Two hidden layers with ReLU activation.
    - Output layer predicts a score for each board position (64 outputs, one for each square).
    - `Adam` optimizer adjusts the model's weights during training.

    - `MSELoss` (Mean Squared Error) is used to measure the difference between predicted values and the target reward.

It uses **ε-greedy exploration** for calculating the value of every move.:

- With probability `exploration_rate`, the AI chooses a random move (exploration).
- Otherwise, it chooses the move with the highest predicted value (exploitation).
- The exploration rate decreases over time using **simulated annealing**.

For each move in the game:

- The target value is the immediate reward plus the discounted future reward (`GAMMA` is the discount factor).
- The model updates its weights using backpropagation to minimise the loss.

The training uses **reinforcement learning** with:

- **Q-value estimation** for each move.
- **Simulated annealing** for exploration-exploitation trade-off.
- **Temporal difference learning** to backpropagate rewards.

Every 100 games, the training progress is logged, showing:

- The model's learning performance (via MSE).
- The outcome of the game.
- Coin counts for both players.

An ANN can be optimised but needs to undergo training again every time, which in this case consists of 10000 games. The optimization would include heuristics or increase the number of layers. Heuristic inclusion is also how many chess-AIs are programmed, this includes logical axioms in the parameters and becomes an implementation of neurosymbolic neural networks,

Convolutional Neural Network

Working of a CNN model:

A Convolutional Neural network (CNN) is a deep learning model that is particularly helpful in analysing spatial data, such as images or grid (Othello can be seen as a

grid.) It works by applying convolutional filters to input data to extract features, pooling layers to reduce dimensionality, and fully connected layers for decision-making. The network learns to identify patterns, such as edges, shapes, or textures, through multiple layers of transformations.

The advantage of using a CNN model to model a game of Othello is that instead of using an ANN, which is a fully connected feed forward neural network, CNN only focuses on a region within its assigned receptive field to make decisions. Hence, the model requires lesser computation- in terms of the weights and biases required for the model, and does not face issues such as overfitting and vanishing gradient problems during training and validation.
Hence, a CNN tends to perform much better than an ANN whenever images/ grid-like data structures are involved.

In the case of Othello, A CNN model works much better than an ANN model since

1. The model's depth can be easily increased without overfitting or gradient problems
2. The game of Othello has a grid-like structure and hence spatial relationships are important to make decisions and devise strategies, therefore, CNN is one of the most suitable models for such a game.

Conclusions

Modern strategies for winning in Othello depend on the strategies that give way to heuristics we used, such as the naive ones: simply maximising the coins you flip each move, or trying to capture the corners because they cannot be converted by the opponent once that happens. The more mathematical strategies build on these ideas, such as assigning weights to each square of the 8x8 grid, depending on the average chance of a capture on that square by the player Black, the average number of coins that get flipped on that square (on average, again this depends on whether the square is accessed early game or left till late game) and how stable that square is. There are even more advanced heuristics such as a combination of stability and mobility, which include a bias towards corners,a bias against all the adjacent squares to the corners (and so on), and keeping track of how many moves are left to a player by making a particular move.

	E-Coins	E-Corners	E-Stability	E-Mobility
Everything	60-4 Everything			
Everything		39-25 Everything		
Everything			47-17 Everything	
Everything				58-6 Everything

Our implementations showed that a simple strategy to capture coins combined with capturing corners whenever possible can beat more in-depth algorithms. ML algorithms beat the need for heuristics, and can simply solve the entire game to find the perfect sequence of moves for an 8x8 implementation, but we have optimised to some extent how to explore the game tree with limited resources. A future idea can be to balance these advanced heuristics by making more stable and passive moves in the beginning, instead of our strategies which allow for more random moves or simply capturing more coins.

References

1. History and Basic rules of http://www.othello.org.hk/tutorials/eng-tu1.html
2. https://www.cs.rochester.edu/~anustup/othello.html
3. “An Analysis of Heuristics in Othello” Vaishnavi Sannidhanam and Muthukaruppan Annamalai, Department of Computer Science and Engineering, University of Washington,
4. M. Buro, The Othello Match of the Year: Takeshi Murakami vs. Logistello , ICCA Journal 20(3), pp 189-193, 1997