CS181 Othello AI

Yinuo Bai, Jiale Lin, Ziwei Shan
ShanghaiTech University
{baiyn2022,linjl2022,shanzw2022}@shanghaitech.edu.cn

Abstract

In this project, we develop multiple game agents for Othello, a board game with complex strategies. We perform extensive optimizations of speed and hyperparameters. Through experiments, we evaluate these agents' performance by analyzing win rates and efficiency. Our results offer insights into each approach and suggest potential refinement for Othello.

Introduction

Othello

Othello is a two-player board game renowned for its elementary rules yet deep strategic complexity. Played on an 8×8 grid, players aim to dominate the board with their discs.

Our group implemented several agents to play Othello using Minimax Search and Approximate Q-learning.

Minimax Search relies on exhaustive exploration of possible moves to determine the optimal strategy. By assuming the opponent plays optimally, the algorithm evaluates the best move. While highly effective, the approach is computationally intensive and its performance heavily depends on the depth of the search.

Approximate Q-learning offers a more adaptive approach allowing the agent to learn through interaction with the game environment. We use both data-based learning and game-based learning. The former leverages historical game data to train the agent. The latter, in contrast, involves the agent learning through games.

Finally, we conduct a series of experiments to evaluate the performance, efficiency, and overall effectiveness of these methods.

Rules of Othello

Objective

The objective of Othello is to have the majority of discs turned to your color by the end of the game.

Setup

Board: Othello is played on an 8×8 square board.

Discs: Each player has 32 discs; one side of each disc is black, and the other side is white.

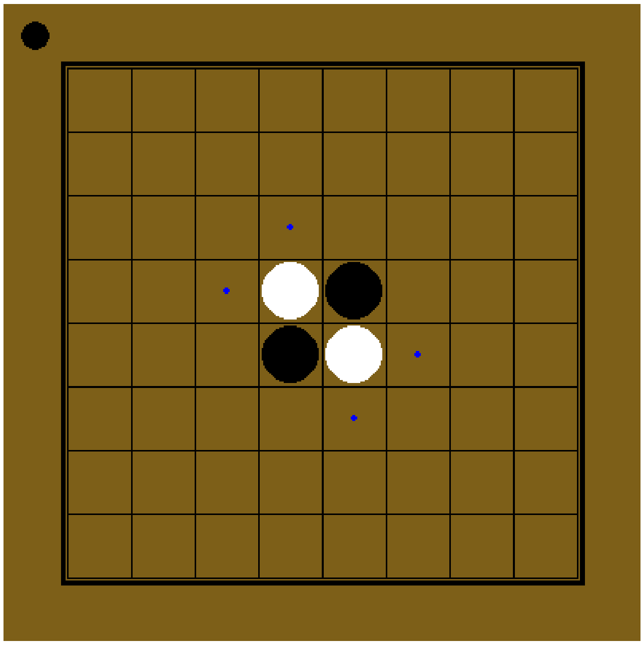


Figure 1: Visualization of the Othello Board

Starting Position: At the beginning of the game, four discs are placed in the center of the board in a square formation. Two black discs are placed on one diagonal, and two white discs are placed on the other diagonal.

Gameplay

Turns: Players take turns, with black typically going first.

Placing Discs: On your turn, place a disc with your color facing up on an empty square.

Capturing Discs: When placing a disc, it must outflank one or more of your opponent's discs. Outflanking means you must place your disc such that there is a straight (horizontal, vertical, or diagonal) line between the disc you place and another disc of your color, with one or more of your opponent's discs in between. All of your opponent's discs that are outflanked are then flipped to your color.

Legal Moves: If a player has no legal moves (i.e., cannot outflank any opponent discs), they must pass their turn. If neither player can move, the game ends.

Copyright © 2024, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Ending the Game
Neither player can make a legal move.
Winning
The player with the majority of their color discs on the board at the end of the game wins.

In case of a tie (both players have the same number of discs), the game is considered a draw.

Methods

Random

In Othello, the Random Agent selects its next move by choosing randomly from all legal placements. Notably, while Random Agents typically underperform in other games, this is not the case in Othello. The game's mechanics, which involve flipping numerous pieces each turn, allow for swift reversals of game situation. Consequently, the Random Agent's performance in Othello is relatively better. The agent's rapid moves, without computations on action choosing, makes it ideal for testing other AI agents and for use in training through gameplays.

Minimax Search

Othello, the two-player, zero-sum, perfect information adversarial game, lends itself to strategic exploration through the application of the minimax algorithm.

Assuming that the adversary employs the optimal strategy, in a situation where our side is to act, the current optimal strategy can be deduced through alternating between the max and min layers of the minimax algorithm. However, due to the vast state space of Othello, with an estimated 10^{28} possible states, we implement a limited search depth and employ alpha-beta pruning to enhance efficiency. Based on prior knowledge(External Resources 3), we design an initial evaluation function and through iterative refinement during experiments, arrive at the following evaluation function:

$$\text{Eval}(s) = \mathbf{W} \cdot \mathbf{H}(s)$$

Here, s denotes a specific state, which includes the state of the board and who the current player is, $\mathbf{W}$ represents a vector of weights, and $\mathbf{H}(s)$ signifies a vector that encapsulates the values of the heuristic function evaluated in the given state s.

Heuristic functions The $\mathbf{H}(s)$ in evaluation function is defined as

$$\mathbf{H}(s) = \begin{bmatrix} H_{\text{pos}}(s) \\ H_{\text{m}}(s) \\ H_{\text{sd}}(s) \\ H_{\text{par}}(s) \end{bmatrix}$$

1. **Position Heuristic Function** Based on the knowledge that the strategy of flipping the most pieces per move is not optimal, and that strategic positions outweigh quantity, we propose position heuristic function. Define the position heuristic function as:

$$H_{pos}(s) = \sum_{i=1}^{N} \sum_{j=1}^{N} X(s)_{i,j} W_{i,j}$$

, where $X(s)$ and W are matrices the same size as the board($N \times N$). $X(s)$ is defined as:

$$X(s) = \begin{cases} 0, & \text{if square (i, j) is empty} \\ 1, & \text{if square (i, j) is occupied by the agent} \\ -1, & \text{if square (i, j) is occupied by the opponent} \end{cases}$$

The weight matrix $\mathbf{W}$ is obtained from the dataset, as detailed in the Data-based Learning section, through statistical analysis. In particular, we calculate $\mathbf{W}$ by subtracting the frequency of moves made by losing players from those made by winners. After considering symmetry, equivalent positions are averaged. The resulting values are then normalized to produce the final weight matrix $\mathbf{W}$.

2. **Mobility Heuristic Function** Recognizing that gaining more maneuverability while restricting your opponent's ability to move effectively can significantly contribute to winning the game. We define $H_{\text{m}}(s)$ as the number of available positions for placing moves within a specific game situation, to represent mobility. This metric captures the potential maneuverability within a given state of the game.

3. **Parity Heuristic Function** The concept of parity enhances the strategic aspect of mobility in Othello. In scenarios without skips (i.e., when a player cannot move), black typically has an even number of empty squares, while white has an odd number. This gives white a subtle edge, as being the last to move can lead to decisive endgame placements. However, this advantage can shift if a player is forced to skip a turn, thereby reversing the parity and potentially favoring the player with odd parity for the win. To capitalize on this strategic element, we introduce the parity heuristic function $H_{\text{par}}(s)$, which serves as an indicator of the parity status based on the current number of empty squares.

4. **Stable Disc Heuristic Function** Stable discs refer to those discs that cannot be flipped anymore, and a higher number of stable discs often indicates a higher likelihood of winning. Therefore, we define $H_{\text{sd}}(s)$ as the count of stable discs. A disc is deemed stable if, in each of the three directions - horizontally, vertically, and diagonally - it is either accompanied by one's own stable discs, at the edge of the board, or surrounded by the opponent's stable discs. This stability is determined recursively, requiring all three directions to satisfy the condition.

Simulated Annealing The $\mathbf{W}$ in evaluation function is defined as

$$\mathbf{W} = \begin{bmatrix} w_{\text{pos}} \\ w_{\text{m}} \\ w_{\text{sd}} \\ w_{\text{par}} \end{bmatrix}$$

To determine the optimal value of $\mathbf{W}$, we apply the Simulated Annealing Algorithm. The flow chart of the algorithm is shown in Figure 2.

During the initialization phase, we randomly assign an initial value to $\mathbf{W}$ and also initialize the current temperature T to an initial temperature value T_0. In each iteration of the loop, a new neighboring weight $\mathbf{W}'$ is generated based

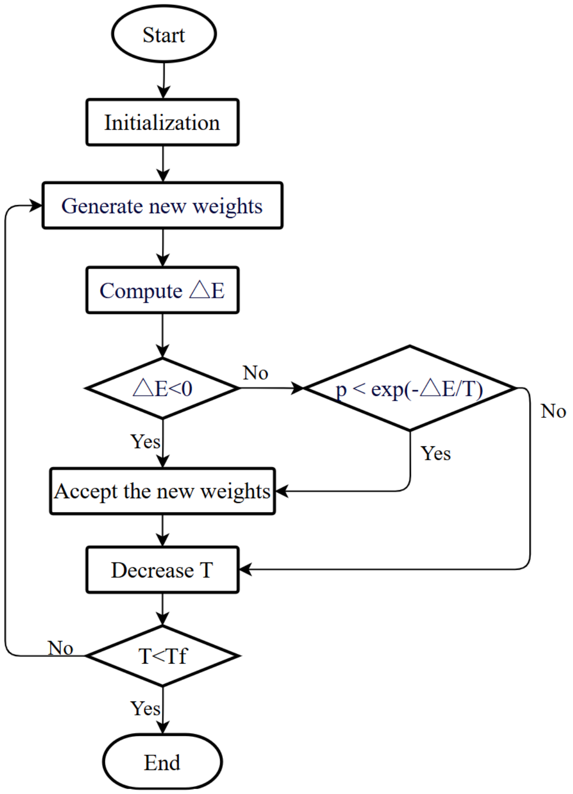


Figure 2: Flow chart of Simulated Annealing Algorithm

on the current temperature T, and the energy difference ΔE is computed. The Metropolis Acceptance Criteria is then applied to determine whether to reject or accept $\mathbf{W}'$. The temperature is gradually reduced until it reaches the final termination temperature T_f.

Here, the calculation of the energy difference ΔE differs from traditional simulated annealing. While we could define the energy by simulating a certain number of games between the Minimax Agent under the new and old weights against an identical agent (such as Random Agent), this approach tends to be time-consuming. To expedite the process, we adopt a more direct method by calculating the difference in the final number of pieces between the games played by the Minimax Agent under the new and old weights, designated as ΔE. This approach avoids the need for calculating the absolute energy and directly yields the desired energy difference.

Approximate Q-learning

As we mentioned before, the state space of Othello is immense. It's impractical to record all states, so we use approximate Q-learning to train our agent.

After various attempts, we have selected some effective features, including the count of discs, the action itself, and features previously used in the Minimax algorithm.

The weight update formula is given by

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha(r + \gamma V(s_{\text{next}}) - \hat{Q}(s_t, a_t))$$

where

$$V(s_{\text{next}}) = \max_a \hat{Q}(s_{\text{next}}, a)$$

Here, s_{next} represents the subsequent state where the same player acts. We estimate the current Q-value using the state encountered after the action.

The learning rate, α, applies an exponential moving average, makes recent moves more important.

r is the reward for the action, which is given by

$$r = \begin{cases} 1, \text{if } s_{\text{next}} \text{ results in a win} \\ -1, \text{if } s_{\text{next}} \text{ results in a loss} \\ 0, \text{otherwise} \end{cases}$$

It's important to note that s_{next} should not be confused with s_{t+2}, which may be assumed as the state after the opponent's move. Our transcript data supports that it's common for one player to run out of moves, allowing the other player to make successive moves, which makes the weight updating more difficult.

To accurately update weights, we log game episodes within our framework and retrospectively adjust the Q-value using the player's previous move.

Data-based learning We downloaded a dataset of 52441 matches from Fédération Française d'Othello (FFO), spanning from 2001 to 2015. These matches represent top-level tournaments across Europe and globally.

The matches data, originally in WTHOR binary format, were converted into a CSV format, providing direct access to details like "tournament name", "transcript", etc. During data cleaning, our focus was on the actions taken by player-agents within these 52441 episodes. We extracted only the transcript data, converting notations like "F6 C3" into actionable pairs for the agents, resulting in a list of action pairs for the entire dataset.

We developed special agents to play Othello based on transcript data. Our data-based learning agent also select moves from this transcript data. The weights are still updated according to our previously given formula. Note that neither of our two Agents makes decisions during the training process.

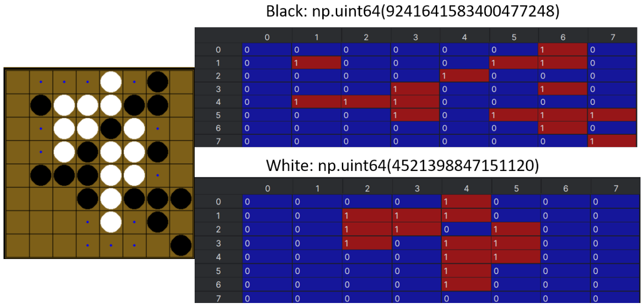
Game-based learning In our game framework, we let Game-based Agent play against Random Agent and Minimax Agent respectively, learning feature weights from thousands of games.

To facilitate exploration in our Q-learning, we implemented an ε-greedy algorithm. We started with an epsilon value of 0.2 and allowed it to decay exponentially over time. This strategy helps to prevent agent to deviate from the optimal combination of weights due to the penalties of random exploration in later training stages. Weights updating still follow callbacks in our framework.

In the subsequent Experiment section, we will provide detailed performance data for our trained Game-based and Data-based Agents according to a series of experiments.

Optimization

This section details the optimization strategies employed in our method. Speed optimization results in quicker responses



Black: np.uint64(9241641583400477248)

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	1	0
1	0	1	0	0	0	1	1	0
2	0	0	0	0	1	0	0	0
3	0	0	0	1	0	0	1	0
4	0	1	1	1	0	0	0	0
5	0	0	0	1	0	1	1	1
6	0	0	0	0	0	0	1	0
7	0	0	0	0	0	0	0	1

White: np.uint64(4521398847151120)

	0	1	2	3	4	5	6	7
0	0	0	0	0	1	0	0	0
1	0	0	1	1	1	0	0	0
2	0	0	1	1	0	1	0	0
3	0	0	1	0	1	1	0	0
4	0	0	0	0	1	1	0	0
5	0	0	0	0	1	0	0	0
6	0	0	0	0	1	0	0	0
7	0	0	0	0	0	0	0	0

Figure 3: Illustration of state aggregation for the chess board

from the Minimax algorithm, accelerated learning via game-based Q-learning, and an overall improved gaming experience for players.

States aggregation leads to a smaller search space and simplifies the recording of game processes.

State Aggregation

A standard Othello board, with 64 squares, can be depicted as an 8×8 matrix where each board represents a black or white disc, or is left empty. During a mid-game two-layers Minimax search, each player's ten-plus possible moves can yield over 10000 successors. Thus, substituting the matrix with a bit-state representation of pieces is beneficial.

We define a new board state as a tuple $(\mathrm{S_B},\ \mathrm{S_W})$, where both $\mathrm{S_B}$ and $\mathrm{S_W}$ represent unsigned 64-bits integers. The 64 squares of the board are sequentially labeled from 0 to 63, starting from the top-left and moving to the bottom-right.

For any square k, if it contains a black disc, the k-th bit of $\mathrm{S_B}$ is set to 1. Similarly, if a square k contains a white disc, the k-th bit of $\mathrm{S_W}$ is set to 1. All unset bits default to 0.

Calculating Successors

As previously discussed, generating successors is a computationally demanding task for the Minimax algorithm. To optimize this, we propose the utilization of bitwise operations on compressed state pairs to efficiently produce new successors.

A successor of the state pair $(\mathrm{S_B},\ \mathrm{S_W})$ is defined by a legal move a, resulting in a new state pair $(\mathrm{S'_B},\ \mathrm{S'_W})$. The process of generating successors, assuming the current player is black, is outlined below:

1. **Pre-judging legality** We examine all 64 squares for potential legal moves. A square k is considered only if it is empty and adjacent to an opponent's disc. A neighbor mask is applied to the opponent's state $\mathrm{S_W}$ for rapid legality assessment.

2. **Status Update** The Othello board's status is updated along diagonals, rows, or columns. Each of the four lines is updated independently, represented by an 8-bit unsigned integer tuple $(\mathrm{L_B},\ \mathrm{L_W})$. Function takes the existing line features as input and output new line features. By querying this pre-calculated function table, we extract the board line features, obtain the new line features, and incorporate them into the new inherited state, thereby eliminating redundant searches during updates.

3. **Feature Extraction** To extract a board line feature, line masks are used to isolate the required 8 or fewer bits from the 64-bit integer. Post bitwise operation with a 64-bit mask, the result remains a 64-bit integer. Subsequently, we consult a pre-processed dictionary (hash table) to swiftly map the 64-bit integer to an 8-bit line feature. The reverse mapping from an 8-bit line feature to a 64-bit update is executed similarly.

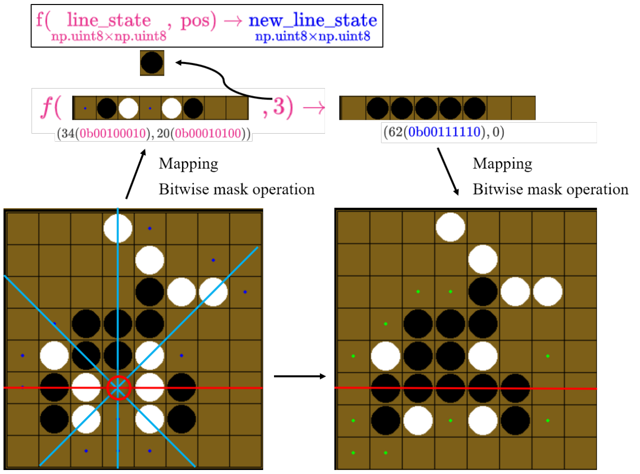


Figure 4: Illustration of successors generating, highlights the update mechanism of a line state, represented by a red line

Cache

In the Minimax tree search, duplicate nodes frequently occur across various steps. Our game framework often requests successors for identical state pairs. To address this, we've implemented a cache using a 150k-entry dictionary, using state pairs as keys. This cache stores all legal moves and their resulting states for a key. When a request for successors corresponds to a previously resolved state pair, we efficiently retrieve the data from the cache using the key. This hash table-based approach significantly accelerates the search process by eliminating redundant calculations.

Experiments

We have conducted a series of experiments to evaluate the performance of different agents, including the Minimax Agent, Data-Based Q-learning Agent, Minimax-Based Q-learning Agent and Random-Based Q-learning Agent. The result are presented in Table 1 and Table 2.

"Minimax(1)" represents a Minimax Search Agent with one min layer and one max layer, while Minimax(2) denotes a Minimax Search Agent with two min layers and two max layers. The Data-based Q-Learning Agent was trained on transcript data for 20000 epochs, the Minimax-based Q-learning Agent was trained against Minimax(1) for 10000 epochs and Random-based Q-Learning Agent was trained against a Random Agent for 10000 epochs.

Winning Rate between different agents

We assess winning rate by launching games for the agents against each other. However, except for experiments involving Random Agents, we face the challenge of not being able to conduct repeated tests on some agents. Since the Minimax Agent consistently responds the same way to other agents, such as our learning agents, which makes decisions based on fixed trained weights, the entire game process is deterministic from the start.

To solve this, we introduce artificial perturbations to both agents, allowing them a low but identical probability of acting randomly. We test different pairs of agents over 200 games, except for Minimax(2). Due to its slow decision-making process, we limit the testing to 100 games when paired with other agents.

Winning Rate against Human

We also let eight students play against different agents, collecting results of 34 games for each agent type to assess their performance against human opponents. The data presents in Table 2 indicates that our Othello agents exhibit awesome performance when playing against humans. Notably, Minimax(2), after SA-optimized, stands out as one of the top performers. The Game-Based Q-learning Agents tend to adopt patterns that can be predicted by humans, whereas the Data-Based Q-learning Agent, which utilizes human tournament transcripts, shows an ability to counter human strategies.

Simulated Annealing

Figure 5 depicts the evolution of **W** over the number of iterations during the simulated annealing algorithm's execution. As the iterations accumulate, it becomes evident that **W** gradually attains a stable state. It is noteworthy that the resulting weight for mobility emerges as a negative value, defying intuitive expectations. This anomalous result possibly stems from the disparity in the gains provided by mobility across various stages of the game.

After experimentation, it has been demonstrated that the new weight assignments are capable of outperforming the Minimax Agent utilizing the previous weight configurations.

Speed Test

We conduct a speed test where Minimax(2) competed against Minimax(1). To evaluate the impact of cache, we also performed multiple trials. The results indicate that an optimized episode averages 9.78 seconds to complete, while a non-optimized episode averages 77.05 seconds. This yields a speedup ratio of 7.88x, significantly enhancing the efficiency of our trains and tests.

External Resources

1. CS181 Programming Assignment Frameworks: Refer to the framework of minimax and reinforcement learning(few references, lots of modifications);

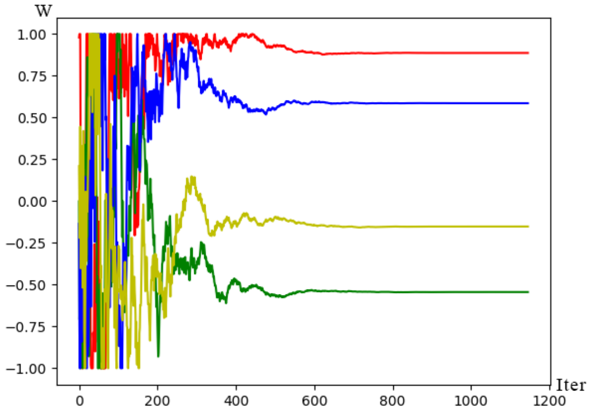


Figure 5: Experiment of Simulated Annealing Algorithm (red: w_{pos}, blue: w_{sd}, yellow: w_{par}, green: w_{m})

2. Dataset from FFO(https://www.ffothello.org/informatique/la-base-wthor/): Dataset;
3. A LA DECOUVERTE D'OTHELLO (https://www.ffothello.org/documents/Livret.pdf): Sources of prior knowledge;
4. PyGame: For visual interface;
5. Numpy: For bitwise operation optimization;
6. Pandas: For dataset processing.

	Minimax(2)	Data-Based Q-learning	Minimax-Based Q-learning	Random-Based Q-Learning
Random	100%	94%	85%	91%
Minimax(1)	100%	32%	55%	24%

Table 1: Winning rate between different agents

	Minimax(2)	Game Based Q-Learning	Data Based Q-Learning
Human	97.0%	70.5%	88.2%

Table 2: Winning Rate Against Human