

# OTHELLO IS SOLVED

Hiroki Takizawa

Preferred Networks, Inc.

Chiyoda-ku, Tokyo, Japan

contact@hiroki-takizawa.name

## ABSTRACT

The game of Othello is one of the world’s most complex and popular games that has yet to be computationally solved. Othello has roughly ten octodecillion (10 to the 58th power) possible game records and ten octillion (10 to the 28th power) possible game positions. The challenge of solving Othello, determining the outcome of a game with no mistake made by either player, has long been a grand challenge in computer science. This paper announces a significant milestone: Othello is now solved. It is computationally proved that perfect play by both players lead to a draw. Strong Othello software has long been built using heuristically designed search techniques. Solving a game provides a solution that enables the software to play the game perfectly.

**Keywords** Othello · Reversi · Games · alpha-beta search

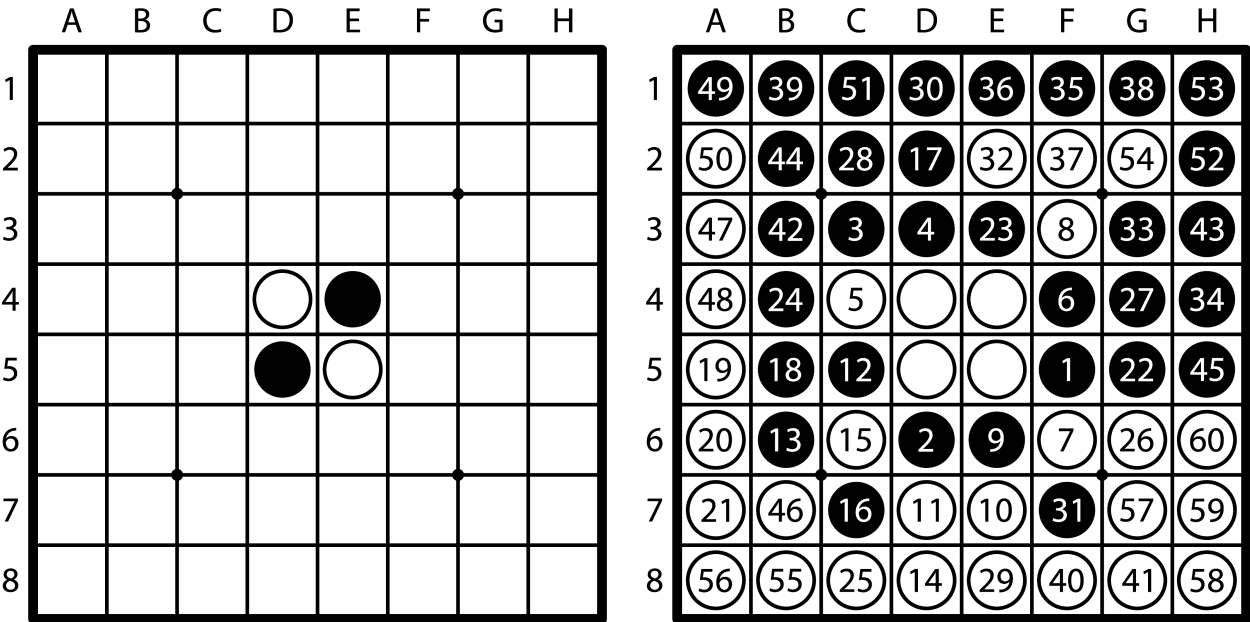


Figure 1: (Left) The initial board position of 8 x 8 Othello. (Right) A diagram of an optimal game record designated by our study. The game record is “F5D6C3D3 C4F4F6F3 E6E7D7C5 B6D8C6C7 D2B5A5A6 A7G5E3B4 C8G6G4C2 E8D1F7E2 G3H4F1E1 F2G1B1F8 G8B3H3B2 H5B7A3A4 A1A2C1H2 H1G2B8A8 G7H8H7H6”. The numbers in the stones indicate the order of moves, and the colors of stones indicate the final result. Our study confirms that if a deviation from this record occurs at any point, our software, playing as the opponent, is guaranteed a draw or win.

# 1 Introduction

Mastering pure strategy games like chess has been considered a symbol of human intelligence. Since the dawn of computer science, this has been a subject of artificial intelligence (AI) research. For example, there were early consideration by Charles Babbage [1] and Claude Shannon [2]. To date, with the enhancement of machine learning techniques and computing capabilities, superhuman-strength software has been developed for some of the most popular games, including chess [3], Go [4], Shogi (Japanese chess) [5, 6], and Othello [7]. However, these superhuman-strength programs cannot perfectly solve the games.

Perfectly solving these games (called games of perfect information) means to determine the final result, which is the outcome of the game under perfect play by both players; this result is termed the “game-theoretic value”. Solved games are classified into at least three types [8, 9]. The most basic type is called ultra-weakly solved games. In this category, we know the game-theoretic value of the initial board position but not any actual winning strategy. Next, in the case of weakly solved games, we know not only the game-theoretic value of the initial position but also a strategy for both players to achieve this value from the initial position under reasonable computational resources. For example, checkers was weakly solved in this sense [10]. At more comprehensive level, we have strongly solved games, where the outcomes are calculated for all possible position that might arise during game-play.

Othello (also called Reversi) is a highly popular game due to its deep strategic nature. It was invented in the 19th century in England, and in the 20th century, the current format of Othello became widespread in Japan and is now played all over the world. The annual World Championships have been held since 1977, which demonstrates its widespread appeal across the globe.

One of the reasons for Othello’s strategic richness is its vast exploration space. Assuming that there are approximately 10 average moves in each position of the game and an overall average of 58 moves throughout a game, the total number of possible game records can be estimated to be around  $10^{58}$ , and the total number of possible board positions was also estimated to be around  $10^{28}$  [8]. These values are far larger than those of any game that has been solved as a grand challenge up to now, including checkers [10]. As a result, Othello has remained unsolved.

In this paper, we announce that we have weakly solved Othello ( $8 \times 8$  board). The game-theoretic value of the initial position turned out to be a draw (an optimal game record and the final result are shown in Figure 1). This is not surprising because human Othello experts already predicted it. Another notable point is that the number of positions we needed to explore to get the strict solution was far less than predicted in previous research[8]. We believe this is due to our sophisticated search algorithm configuration.

The Othello result is a monumental achievement for humanity, demonstrating the remarkable advances in computer science and AI technology. Solving Othello has been one of the grand challenges for AI. Over recent decades, AI capabilities have expanded owing to advances in both computing power and algorithms, including enhanced search techniques. In our study, even with the use of the latest computer cluster, solving Othello remained a significant hurdle. Our breakthrough came by improving search efficiency and modifying the latest Othello software.

This paper describes our method to solve Othello, several findings as results, and implications of this research. The raw data and programs to reproduce the results are available on GitHub, Zenodo, and figshare (see Data Availability section).

## 2 Related Works

### 2.1 Solved Games

To the best of our knowledge, the latest game solved prior to this study as a grand challenge is checkers [10]. However, many nontrivial games have been solved, including Connect Four [8], Qubic [8], Go-Moku [8], Nine Men’s Morris [11], and Awari [12]. The difficulty of solving these games largely depends on the number of positions or situations in the game. Solving a game not only reveals its outcome but can also be useful in creating puzzles based on that game [9].

### 2.2 Solving Technique

The algorithms used to solve games have been extensively studied, and they are chosen based on the purpose and nature of the game. For weak solutions, alpha-beta search [13] is often used, while retrograde analysis [14] is frequently used for strong solutions. Additionally, algorithms like depth-first proof-number (df-pn) search [15, 16], which is based on proof-number (pn) search [17], have been developed to solve puzzles with very long solution sequences. In our study, we utilized alpha-beta search because our goal was to obtain a weak solution.

## 2.3 Algorithms for Parallel Search

Alpha-beta search is an algorithm that sequentially performs depth-first search of a game graph, and naive parallelization does not improve search efficiency very much. Many algorithms have been developed for efficient parallelization. Young Brothers Wait Concept (YBWC) [18] and Lazy SMP [19] are popular methods for shared memory environments (e.g., a single computer). Algorithms suited for distributed memory environments (e.g., supercomputers or cloud computing) might include Asynchronous Parallel Hierarchical Iterative Deepening (APHID) [20] and ABDADA [21]. However, distributed memory environments greatly differ due to various factors including the bandwidth and latency of interconnects between nodes, and thus the appropriate algorithms may also differ. Therefore, developers who work with current or future distributed memory environments may need to choose or develop algorithms that are appropriate for those environments.

## 3 Methods

### 3.1 Use of Terms “Ply” and “Move”

In chess, there is a tradition where two sequential moves, one from white and the other from black, are called a “move” or “full-move”, while an individual move is called a “ply” or “half-move”. However, in this article, we refrain from using “ply” and always use “move” to denote an individual move.

### 3.2 Rules of Othello

The rules of Othello are as follows:

1. Black moves first, after which the players alternate.
2. If there is an empty square that satisfies the conditions for placing a stone (see below), the player to move must choose one of such empty squares and place a stone there. If there is no empty square that satisfies the condition, the player must pass their turn.
3. If a player places a stone such that there are opponent’s stones in a straight line (horizontal, vertical, or diagonal) between this new stone and another of the player’s stones, with no empty squares in between, then the opponent’s stones in that line are flipped to become the player’s stones.
4. Each player can only place a stone on the squares where placing a stone flips one or more of the opponent’s stones.
5. The game ends when the board is completely filled or when there is no square on which either player can place a stone.
6. The player with more stones on the board at the end of the game wins.

The difference in the number of stones at the end of the game is called the “score”.

### 3.3 Modification to Edax

Existing Othello software called Edax [22] was used to solve the position with 36 empty squares. Edax is based on alpha-beta search [13] and employs many techniques to improve search efficiency. While Edax is among the strongest software under typical match rules (e.g., 10 seconds per one move), its algorithm was considered suboptimal for the purpose of solving games over tens of minutes in situations where there is a large difference in scores. Therefore, the following two modifications were made:

- We disabled aspiration search during iterative deepening. In other words, when narrowing down alpha and beta for a solving, we modified it to take wider alpha and beta values during the shallow iterations of iterative deepening. This is because there is a tendency for the computation time to increase when the principal variation is updated. Therefore, during shallow iterations, even if a move results in a fail-high, the search should not be terminated, and instead, the best move should be sought.
- When performing move ordering, if the results from a shallow search are found in the transposition table, we ignore them if they are relatively too shallow. This is because, especially in a solving, the search depth becomes very high. In regions where the search depth is shallow, the accuracy of move ordering based on the transposition table greatly impacts performance.

The source code of modified Edax is available on GitHub and Zenodo (see Data Availability section).

### 3.4 Obtaining a set of target positions (with 50 empty squares) by optimal alpha-beta search

---

**Algorithm 1**  $G_{50}(p, D_{50})$ : Generate subset of positions with 50 empty squares as sub-problems.

---

**Require:**  $p$ : A position.

**Require:**  $D_{50}$ : A dictionary where the keys are all positions with 50 empty squares, and the values are their respective predictive scores.

**Ensure:** set of positions such that if all positions in it are solved and all solutions match the predictions, the initial position is consequently solved.

```

1:  $M \leftarrow$  a list of all legal moves on  $p$ 
2: if  $M$  is empty then
3:    $p_{next} \leftarrow$  the position after applying pass to  $p$ 
4:    $M' \leftarrow$  a list of all legal moves on  $p_{next}$ 
5:   if  $M'$  is empty then
6:     return  $\{\}$  ▷ Return an empty set.
7:   end if
8:   return  $G_{50}(p_{next}, D_{50})$ 
9: end if
10: if  $p$  has 50 empty squares then
11:   return  $\{p\}$  ▷ Return a set consisting of only  $p$ .
12: end if
13:  $m, s \leftarrow$  the best move  $m$  and corresponding score  $s$  ▷ Perform another search from  $p$  to positions in  $D_{50}$  to obtain  $m$  and  $s$ .
14: if  $s > 0$  then
15:    $p_{next} \leftarrow$  the position after applying  $m$  to  $p$ 
16:   return  $G_{50}(p_{next}, D_{50})$  ▷ Fail-high always occurs.
17: end if
18:  $l \leftarrow \{\}$  ▷ An empty set.
19: for  $m \in M$  do
20:    $p_{next} \leftarrow$  the position after applying  $m$  to  $p$ 
21:    $l \leftarrow l + G_{50}(p_{next}, D_{50})$  ▷ Fail-high never occurs.
22: end for
23: return  $l$ 

```

---

We developed an algorithm that requires predictive scores for all positions with 50 empty squares and returns a subset such that if all positions belonging to that subset are solved and all solutions match the predictions, the initial position is consequently solved. This is described by Algorithm 1. This algorithm is similar to the alpha-beta search with  $(\alpha, \beta) = (-1, 1)$  fixed. By inputting the initial position as the first argument and the dictionary of the positions and predictive scores as the second, a subset satisfying the conditions can be obtained. Notably, to keep the number of elements in the subset small, this algorithm internally performs the other alpha-beta search. As is commonly known, alpha-beta search is most efficient when the best move is searched first. Therefore, an internal alpha-beta search is conducted to find the best move. This inner search can be made more efficient through elementary memoization. As a result, even when implementing the inner search, it's possible to ensure that the computational time complexity does not increase.

### 3.5 Obtaining a set of target positions (with 36 empty squares) by optimal alpha-beta search

We implemented Algorithm 5, which requires a position with 50 empty squares and data about position(s) with 36 empty squares and outputs a set of position(s) with 36 empty squares and a corresponding result hypothesis. This algorithm can process known search outcomes for positions with 36 empty squares, and output position(s) with 36 empty squares and corresponding estimated game-theoretic value; if we can confirm that all outputted estimations are correct, then we can prove the game-theoretic value of the input position. Importantly, it can differentiate between positions that we have obtained game-theoretic value for and those that we have only estimated the value of.

Algorithms 2, 3, and 4 are auxiliary algorithms for Algorithm 5. These three algorithms can be made more efficient through elementary memoization, which is omitted from the pseudo-codes. The source code is available at GitHub (see Data Availability section).

We solved the positions with 36 empty squares, which were obtained from Algorithm 5, using a computer cluster and Edax software. If the predicted value of the results was less than 30, we read with 4 cores; otherwise, we did so with 1

---

**Algorithm 2**  $E(p, D')$ : Estimate Game-Theoretic Value of the Given Position.

---

**Require:**  $p$ : A position.**Require:**  $D'$ : A dictionary; key is a position and value is an estimation of its game-theoretic value.**Ensure:** An integer that is an estimation of game-theoretic value of  $p$ .

```
1: if  $p \in D'$  then
2:   return  $D'[p]$ 
3: end if
4:  $f \leftarrow$  Edax's static evaluation function
5:  $v \leftarrow f(p)$ 
6: if  $|v| > 10$  then
7:   return  $v$ 
8: end if
9:  $\alpha \leftarrow \min(-3, v)$ 
10:  $\beta \leftarrow \max(3, v)$ 
11: return the value determined by an alpha-beta search from  $p$  to a depth of 2, using  $f$  as the evaluation function for leaf nodes and  $\alpha, \beta$  as initial alpha and beta.
```

---

core. In the initial phase of this computation, we did not have confidence that the game-theoretical value of the initial position would be a draw, so we set the alpha-beta window to  $[-3, +3]$ . Subsequently, when reading with 4 cores, we changed it to read with a  $[-1, +1]$  window. When reading with 1 core, we always kept the window at  $[-3, +3]$ . The advantage of reading with 1 core is avoiding efficiency degradation due to parallel searching, and because the number of search positions becomes deterministic, it ensures reproducibility. The continued benefit of keeping the window at  $[-3, +3]$  is that if the game-theoretical value of the initial position turned out to be -2 rather than a draw, there would be no need for re-computation, and there is no need to change the command-line options of Edax, meaning we do not have to save it for each problem. The downside is a possible slight increase in the number of search positions, but for positions with a significant difference, it is believed to have minimal impact compared to when the window is set to  $[-1, +1]$ .

The game-theoretic values of the output positions can sometimes deviate from the estimations of the above algorithm. In such cases, by adding the results to  $D$  and rerunning the Algorithm 5, more positions to be solved can be identified. Once Algorithm 5 no longer outputs any position, it can be said that the proof of the input position with 50 empty squares has been established. This procedure can be described as Algorithm 6.

### 3.6 Constructing a program that never loses

To satisfy the Weakly solved condition, we implemented a Python script that acts as a perfect (i.e., never loses) player by referring to the results. The script refers to the result table while there are more than 36 empty squares, and after that, it delegates to Edax to play perfectly.

### 3.7 Materials

To solve positions with 36 empty squares, we used the CPUs of MN-J, a supercomputer owned by Preferred Networks Inc. MN-J refers collectively to multiple supercomputers: MN-2A, MN-2B, and MN-3, all of which appear to users as a single Kubernetes cluster. This supercomputer is equipped with several types of CPUs (Intel Xeon 6254, AMD EPYC 7713, Intel Xeon 8380, and Intel Xeon 8260M), and all CPUs feature main memory with Error Checking and Correction (ECC).

---

**Algorithm 3**  $G_{36}^{third}(p, D, k, \alpha, \beta)$ : Calculate an upper or lower bound of Game-Theoretic Value of the Given Position.

---

**Require:**  $p$ : A position.

**Require:**  $D$ : A dictionary; key is a position and value is the exact upper and lower bounds of its game-theoretic value.

**Require:**  $k$ : A Boolean that indicates the kind of search, i.e., upper or lower.

**Require:**  $\alpha, \beta$ : Parameters for alpha-beta search.

**Ensure:** An integer that is an upper (or lower; determined by  $k$ ) bound of the game-theoretic value of  $p$ .

```

1:  $M \leftarrow$  a list of all legal moves on  $p$ .
2: if  $M$  is empty then
3:    $p_{next} \leftarrow$  the position after applying pass to  $p$ 
4:    $M' \leftarrow$  a list of all legal moves on  $p_{next}$ 
5:   if  $M'$  is empty then
6:     return the final value of  $p$ . ▷ The game is ended.
7:   end if
8:   return  $-G_{36}^{third}(p_{next}, D, \neg k, -\beta, -\alpha)$ 
9: end if
10: if the number of empty squares is 36 then
11:   if  $p \in D$  then
12:     if  $k$  then
13:       return  $D[p].upperbound$  ▷ Return an upper bound of game-theoretic value of  $p$ .
14:     end if
15:     return  $D[p].lowerbound$  ▷ Return an lower bound of game-theoretic value of  $p$ .
16:   end if
17:   if there is a move that causes wipe-out then
18:     return 64
19:   end if
20:   return 64
21: end if
22: Sort  $M$  in descending order of how promising the result is using a deterministic method.
23:  $v \leftarrow$  an empty list.
24: for  $m \in M$  do
25:    $p_{next} \leftarrow$  the position after applying  $m$  to  $p$ 
26:    $v \leftarrow v + -G_{36}^{third}(p_{next}, D, \neg k, -\beta, -\alpha)$  ▷ Nega-max search. Nega-scout can be used.
27:   if  $\max(v) \geq \beta$  then
28:     return  $\max(v)$  ▷ Fail-high occurred.
29:   end if
30: end for
31: return  $\max(v)$ 

```

---

---

**Algorithm 4**  $G_{36}^{second}(p, D, D', \alpha, \beta)$ : Calculate an estimation of Game-Theoretic Value of the Given Position.

---

**Require:**  $p$ : A position.

**Require:**  $D$ : A dictionary; key is a position and value is the exact upper and lower bounds of its game-theoretic value.

**Require:**  $D'$ : A dictionary; key is a position and value is an estimation of its game-theoretic value.

**Require:**  $\alpha, \beta$ : Parameters for alpha-beta search.

**Ensure:** An integer that is an estimation of game-theoretic value of  $p$ .

```

1:  $M \leftarrow$  a list of all legal moves on  $p$ .
2: if  $M$  is empty then
3:    $p_{next} \leftarrow$  the position after applying pass to  $p$ 
4:    $M' \leftarrow$  a list of all legal moves on  $p_{next}$ 
5:   if  $M'$  is empty then
6:     return the final value of  $p$ . ▷ The game is ended.
7:   end if
8:   return  $-G_{36}^{second}(p_{next}, D, D', -\beta, -\alpha)$ 
9: end if
10: if the number of empty squares is 36 then
11:   if there is a move that causes wipe-out then
12:     return 64
13:   end if
14:    $e \leftarrow E(p, D')$  ▷ Call algorithm 2.
15:   if  $p \in D$  then
16:     if  $D[p].lowerbound = D[p].upperbound$  then
17:       return  $D[p].upperbound$  ▷ Return an exact game-theoretic value of  $p$ .
18:     else if  $\min(\beta, e) \leq D[p].lowerbound$  then
19:       return  $D[p].lowerbound$  ▷ Return a lower bound of game-theoretic value of  $p$ .
20:     else if  $D[p].upperbound \leq \max(\alpha, e)$  then
21:       return  $D[p].upperbound$  ▷ Return an upper bound of game-theoretic value of  $p$ .
22:     end if
23:     return  $e$  ▷ Return an estimation of game-theoretic value of  $p$ .
24:   end if
25:   return 64
26: end if
27: Sort  $M$  in descending order of how promising by a deterministic method.
28:  $v \leftarrow$  an empty list.
29: for  $m \in M$  do
30:    $p_{next} \leftarrow$  the position after applying  $m$  to  $p$ 
31:    $v \leftarrow v + -G_{36}^{second}(p_{next}, D, D', -\beta, -\alpha)$  ▷ Nega-max search. Nega-scout can be used.
32:   if  $\max(v) \geq \beta$  then
33:     return  $\max(v)$  ▷ Fail-high occurred.
34:   end if
35: end for
36: return  $\max(v)$ 

```

---

---

**Algorithm 5**  $G_{36}^{first}(p, D, D', A, \alpha, \beta)$ : Traverse Game-Graph and Obtain Positions to Solve.

---

**Require:**  $p, D, D', \alpha, \beta$ : The same parameters as in Algorithm 4.

**Require:**  $A$ : A dictionary in which key is a position with 36 empty square and value is a tuple; each tuple consists of an estimated game-theoretic value and two integers  $\alpha$  and  $\beta$ . To solve  $p$ , we should solve all positions in the key under the  $\alpha$  and  $\beta$  in the value. If all estimations are correct, then  $p$  is solved.

**Ensure:** An integer that is an estimation of game-theoretic value of  $p$ .

**Ensure:** The updated  $A$ .

```

1:  $M \leftarrow$  a list of all legal moves on  $p$ .
2: if  $M$  is empty then
3:    $p_{next} \leftarrow$  the position after applying pass to  $p$ 
4:    $M' \leftarrow$  a list of all legal moves on  $p_{next}$ 
5:   if  $M'$  is empty then
6:     return the final value of  $p$ , and  $A$ . ▷ The game is ended.
7:   end if
8:   return  $-G_{36}^{first}(p, D, D', A, -\beta, -\alpha)$ 
9: end if
10: if the number of empty squares is 36 then
11:   if there is a move that causes wipe-out then
12:     return 64,  $A$ 
13:   end if
14:    $e \leftarrow G_{36}^{second}(p, D, D', \alpha, \beta)$ 
15:   if  $p \notin D$  then
16:      $A[p] \leftarrow (\alpha, \beta, e)$ 
17:   else
18:      $A[p] \leftarrow (\min(\alpha, A[p].\alpha), \max(\beta, A[p].\beta), e)$ 
19:   end if
20:   return  $e, A$  ▷ Return an estimation of game-theoretic value of  $p$ .
21: end if
22:  $v \leftarrow$  an empty list.
23:  $S \leftarrow$  an empty dictionary.
24: for  $m \in M$  do
25:    $p_{next} \leftarrow$  the position after applying  $m$  to  $p$ 
26:    $e_{lower} \leftarrow -G_{36}^{third}(p_{next}, D, True, -64, 64)$  ▷ True means upper-mode.
27:    $e_{upper} \leftarrow -G_{36}^{third}(p_{next}, D, False, -64, 64)$  ▷ False means lower-mode.
28:   if  $\beta \leq e_{lower}$  then ▷ Fail-high occurred.
29:     return  $e_{lower}, A$ 
30:   else if  $e_{upper} \leq \alpha$  then
31:      $v \leftarrow v + e_{upper}$ 
32:     remove  $m$  from  $M$ .
33:     continue.
34:   else if  $\alpha < e_{lower} = e_{upper} < \beta$  then
35:      $v \leftarrow v + e_{upper}$ 
36:      $\alpha \leftarrow e_{upper}$ 
37:     remove  $m$  from  $M$ .
38:     continue.
39:   end if
40:    $e \leftarrow -G_{36}^{second}(p_{next}, D, D', -64, 64)$ 
41:    $S[m] \leftarrow e$  ▷ We can add  $S[m]$  to some auxiliary heuristic factors to improve the following move ordering.
42: end for
43: if  $M$  is empty then
44:   return  $\max(v), A$ 
45: end if
46: Sort  $M$  in descending order of the values in  $S$ .
47: for  $m \in M$  do
48:   Perform the same alpha-beta search as in Algorithms 3 and 4.
49: end for
50: return  $\max(v), A$ 

```

---



---

**Algorithm 6**  $Q(p, \alpha, \beta)$ : Calculate the Game-Theoretic Value of the Given Position.

---

**Require:**  $p$ : A position.

**Require:**  $\alpha, \beta$ : Parameters for alpha-beta search.

**Ensure:** An integer that is the game-theoretic value of  $p$ .

**Ensure:** A dictionary; key is a position and value is the exact upper and lower bounds of its game-theoretic value. One can prove  $p$  only from the information in this dictionary.

```
1:  $D \leftarrow$  an empty dictionary.
2:  $D' \leftarrow$  an empty dictionary.
3: while  $True$  do
4:    $A \leftarrow$  an empty dictionary.
5:    $v, A \leftarrow G_{36}^{first}(p, D.D', A, \alpha, \beta)$ 
6:   if  $A$  is empty then
7:     return  $v, D$ 
8:   end if
9:    $R \leftarrow$  The game-theoretic values (or its estimation) of all positions in  $A$ . ▷ Edax is available.
10:   $D \leftarrow D +$  The exact game-theoretic values in  $R$ .
11:   $D' \leftarrow D' +$  The estimations in  $R$ .
12: end while
```

---

## 4 Results

First of all, we enumerated and shortly evaluated all positions with 50 empty squares. We only enumerated positions with at least one legal move and considered symmetrical positions to be identical. As a result, 2,958,551 positions were enumerated. We evaluated all of them by Edax for 10 seconds using a single CPU core. For positions that resulted in values close to a draw from the 10-second evaluations, we conducted more extended evaluations.

Next, we selected 2,587 positions out of the 2,958,551 positions and formulated hypotheses regarding their game-theoretic values. We chose them such that if all these hypotheses were proven correct, it would prove that the initial position results in a draw. Although there are numerous ways to select subsets that would prove that the initial position results in a draw, we used Algorithm 1 to obtain a small subset. For the evaluation values, we used the values obtained from the previously mentioned evaluations. In cases where the values were the same, we prioritized positions that appear frequently in the WTHOR database[23] of Othello games published by the French Othello Federation. We used a dataset including 61,549 game records played between 2001 and 2020. As we will describe in detail later, it was proven that all these 2,587 hypotheses were correct.

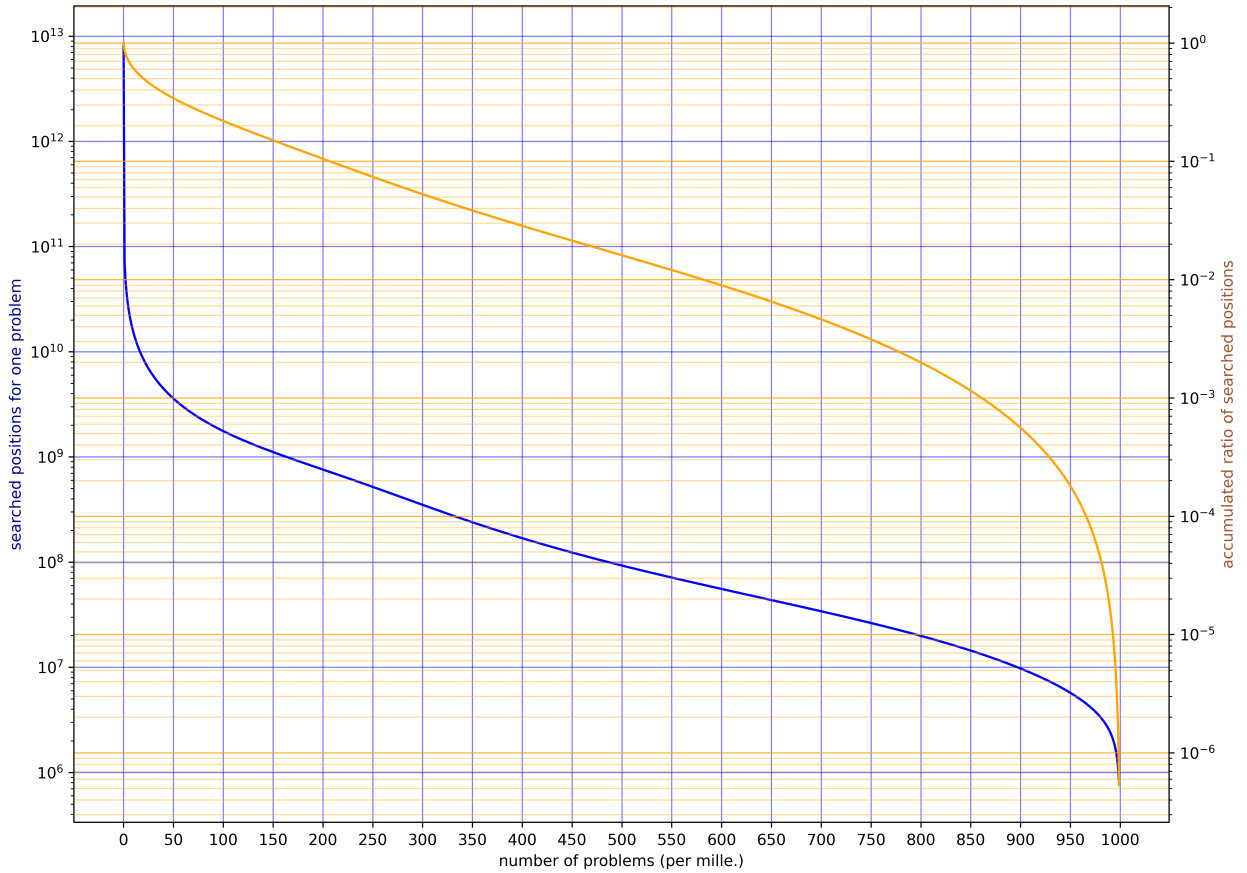


Figure 2: Positions with 36 empty squares that were solved to prove the initial position were sorted in descending order by the number of search phases reported by Edax, and the cumulative number (orange) and number of search phases for a single particular question (blue) were plotted for each 1/1000.

As a result, the number of positions with 36 empty squares needed to solve the initial position amounted to 1,505,367,525, with the total search positions reported by Edax for all these positions reaching 1,526,001,455,595,489,506 (Figure 2, Table 1). Due to having the alpha-beta window set to  $[-3, +3]$  for some borderline positions, there seems to be room for reduction in this number. As a null-window search is available for verification, the number of necessary search positions could potentially be even lower.

The results of the opening are illustrated in Figure 4. Our perfect player never voluntarily deviates from the optimal game record (shown as bold black moves). If the opponent chooses a move not shown in this figure, we proved that our

Table 1: Problem size and search capability

Factor	Description	Number
Problems	Number of solved problems to weakly solve Othello	$\sim 1.5 \times 10^9$
CPU	Searching capability using Edax (positions/GHz/core/sec)	$\sim 1.2 \times 10^7$
Positions	Number of searched positions (reported by Edax)	$\sim 1.5 \times 10^{18}$

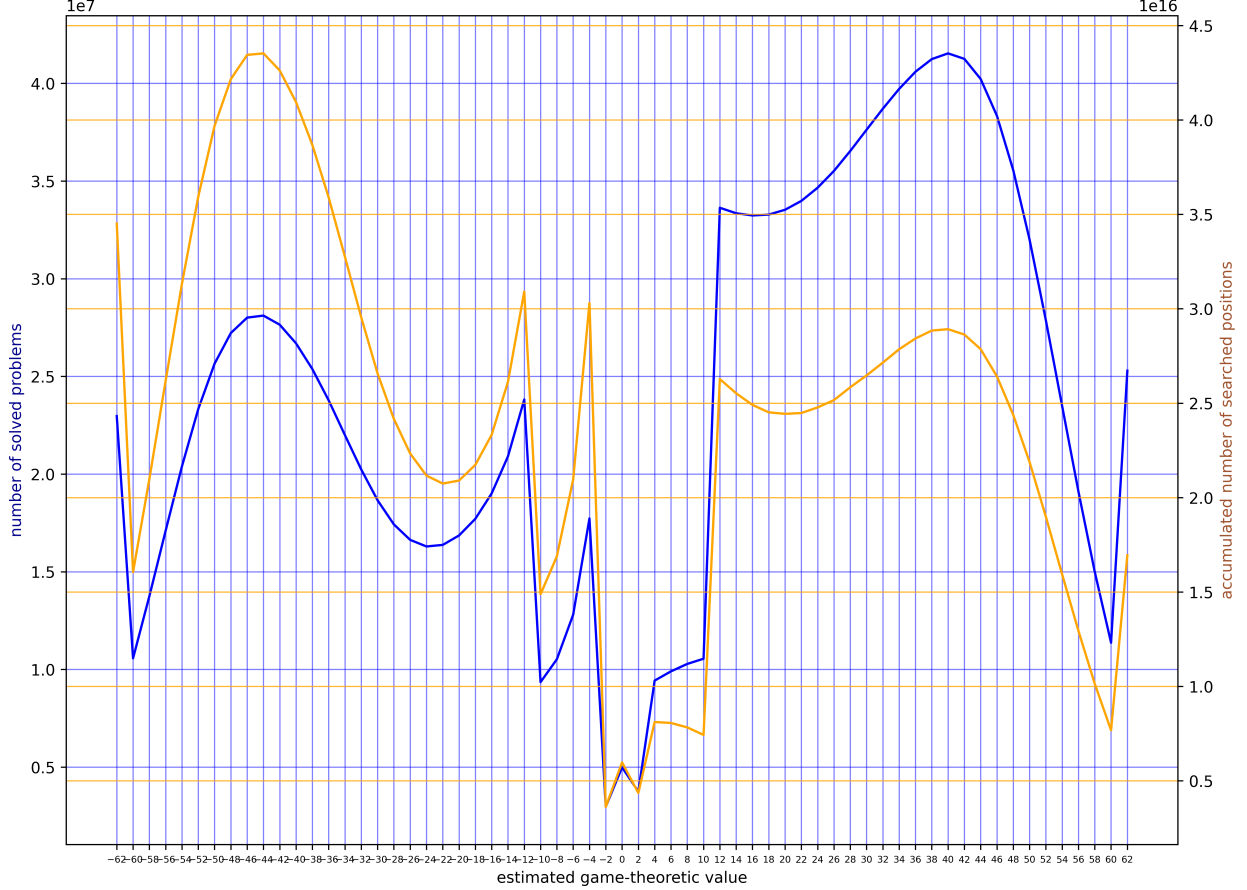


Figure 3: Positions with 36 empty squares that were solved to prove the initial position were classified according to the value of Algorithm 2 (horizontal axis) and the sum of the numbers of searched positions reported by Edax was calculated for each (vertical axis).

player will always win. If the opponent chooses one of the non-bold moves, we proved that our player draws or wins by choosing moves shown as bold gray moves.

## 5 Discussion and Conclusions

We conclude that our study has weakly solved Othello, although we recognize that our achievement is just above the criteria for weakly solving. For certain borderline positions with 36 empty squares, Edax requires a large amount of computation to determine the game-theoretic value and corresponding move. However, given the continuing advances in personal computers, it is reasonable to conclude that our approach requires only reasonable computational resources. By providing an additional “opening” book for these positions with 35 or fewer empty squares, we could further reduce the computational demand. However, to expedite our announcement, we opted against computing additional books in this study. Nonetheless, there may be interest among Othello enthusiasts for software that can determine the best move using fewer computational resources.

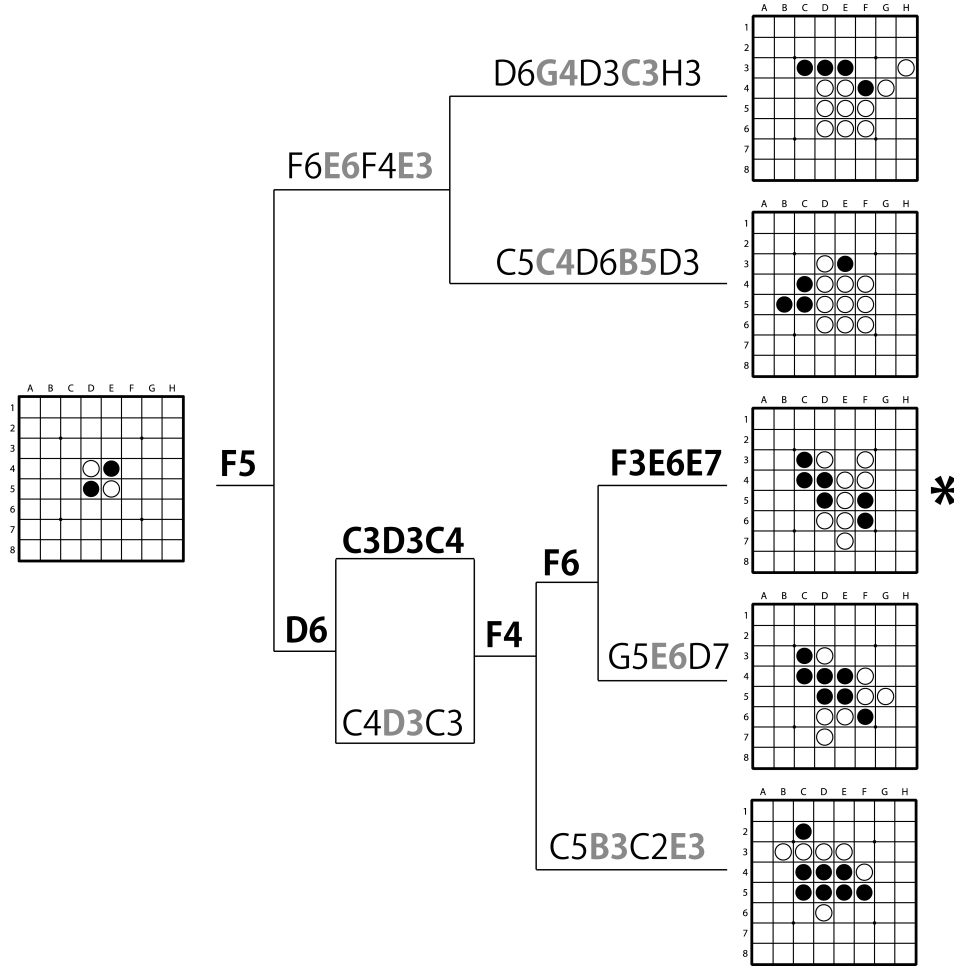


Figure 4: A graphical representation of results about opening of Othello. The bold black moves show the optimal game record. Our perfect player always chooses the bold (black or gray) moves in the corresponding position. Right five positions are proved that all those game-theoretic values are draws. Center one with asterisk is the progress of Figure 1.

As Figure 3 indicates, many of our calculations to weakly solve Othello were devoted to positions where, according to the estimation, there is a clear advantage in terms of winning or losing. This indicates that one cannot claim a pseudo-solution by not proving positions whose estimated game-theoretic value exceeds any threshold.

As Figure 3 implies, for positions that were expected to have a significant difference in scores, there were some in which a solving by Edax took a large amount of time. It is possible that the systematic and significant errors in Edax's function to estimate the game-theoretic value from a position (called the static evaluation function), especially for positions unlikely to appear in actual games, are the reason for this. Regarding this issue, while it can be addressed by preparing an additional opening book, there is also potential for retraining or improving the design of the static evaluation function.

We recognize that some readers may be skeptical about the validity of computational proofs. Naturally, computational errors due to CPU or memory faults cannot be entirely ruled out. However, as the vast majority of calculations were executed on a computer cluster with ECC memory, we believe the results to be nearly indisputable. Moreover, even if a computational error were present, the chance of overturning our conclusion of a final draw is extremely low. If any errors are detected, they can be easily recalculated using the publicly released software.

To the best of our knowledge, no category in between weakly and strongly solving has been proposed. We considered strongly solving Othello is intractable and aimed for a weak solution. We have created software that will always achieve a draw or win to achieve the criteria for weakly solving. If the opponent makes a blunder, however, we do not guarantee that the software capitalizes on it.

Although strongly solving the game may be intractable, developing software that consistently makes the best move represents a challenge that lies between weak and strong solving, and is likely to attract widespread interest. Therefore, we would propose to call this intermediate category "semi-strong solving". This study does not achieve semi-strong solving of Othello; this remains as future work.

Considering the game's popularity and estimated size of the search space, we speculate that chess might be the next weakly solved grand challenge. However, because the search space of chess is very large, not only improvements in computational power but also theoretical breakthroughs might be necessary. We hope that this study will inspire readers and contribute to significant advancements in future computer science.

## Acknowledgments

The author gratefully thanks members of Preferred Networks Inc., including Dr. Kenta Oono, Dr. Kohei Hayashi, Dr. Masanori Koyama, and Dr. Shin-ichi Maeda, for their useful discussions and encouragement.

Parts of this study, especially the majority of the calculations, were conducted during the author's work time at Preferred Networks Inc. This was made possible by the company's 20-percent rule, which allows employees to dedicate 20 percent of their work time to pursue their own ideas and projects. The author gratefully thanks the company for having the rule.

## Additional Information and Declarations

### Competing Interests

The author declares that there are no competing interests.

### Author Contributions

Hiroki Takizawa conceived and designed the research, implemented and performed the computational experiments, analyzed the data, prepared figures and tables, authored drafts of the paper, and approved the final draft.

### Data Availability

The source code of modified Edax is available at GitHub (<https://github.com/eukaryo/edax-reversi-AVX-v446mod2>), and Zenodo (<https://doi.org/10.5281/zenodo.10030906>).

The raw outputs of analyses are available at figshare (<https://doi.org/10.6084/m9.figshare.24420619>).

The source code for analyses are available at GitHub (<https://github.com/eukaryo/reversi-scripts>).

### Funding

The author did not receive any academic funding for this study.

## References

- [1] Charles Babbage. *Passages from the Life of a Philosopher*. London: Longman, 1864.
- [2] Claude E Shannon. Xxii. programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(314):256–275, 1950.
- [3] Murray Campbell, A. Joseph Hoane, and Feng-hsiung Hsu. Deep blue. *Artif. Intell.*, 134(1–2):57–83, January 2002.
- [4] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.
- [5] Tomoyuki Kaneko and Takenobu Takizawa. Computer shogi tournaments and techniques. *IEEE Transactions on Games*, 11(3):267–274, 2019.

- [6] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dhharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.
- [7] Michael Buro. The othello match of the year: Takeshi murakami vs. logistello. *ICGA Journal*, 20(3):189–193, 1997.
- [8] L. V. Allis. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, Department of Computer Science, University of Limburg, 1994.
- [9] Hiroki Takizawa. Strongly solved ostle: calculating a strong solution helps compose high-quality puzzles for recent games. *PeerJ Computer Science*, 9:e1560, 2023.
- [10] Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *science*, 317(5844):1518–1522, 2007.
- [11] Ralph Gasser. Solving nine men’s morris. *Computational Intelligence*, 12(1):24–41, 1996.
- [12] John W Romein and Henri E Bal. Solving awari with parallel retrograde analysis. *Computer*, 36(10):26–33, 2003.
- [13] Donald E Knuth and Ronald W Moore. An analysis of alpha-beta pruning. *Artificial intelligence*, 6(4):293–326, 1975.
- [14] Ken Thompson. Retrograde analysis of certain endgames. *J. Int. Comput. Games Assoc.*, 9(3):131–139, 1986.
- [15] Ayumu Nagai. *Df-pn algorithm for searching AND/OR trees and its applications*. PhD thesis, Department of Information Science, University of Tokyo, 2002.
- [16] Akihiro Kishimoto, Mark HM Winands, Martin Müller, and Jahn-Takeshi Saito. Game-tree search using proof numbers: The first twenty years. *Icga Journal*, 35(3):131–156, 2012.
- [17] L.Victor Allis, Maarten van der Meulen, and H.Jaap van den Herik. Proof-number search. *Artificial Intelligence*, 66(1):91–124, 1994.
- [18] Rainer Feldmann, Burkhard Monien, Peter Mysliwietz, and Oliver Vornberger. Distributed game tree search. *Parallel Algorithms for Machine Intelligence and Vision*, pages 66–101, 1990.
- [19] Emil Fredrik Østensen. A complete chess engine parallelized using lazy smp. Master’s thesis, University of Oslo, 2016.
- [20] Mark Gordon Brockington. *Asynchronous Parallel Garne-Tree Search*. PhD thesis, University of Alberta, 1998.
- [21] Jean-Christophe Weill. The abdada distributed minimax search algorithm. In *Proceedings of the 1996 ACM 24th Annual Conference on Computer Science, CSC ’96*, page 131–138, New York, NY, USA, 1996. Association for Computing Machinery.
- [22] Richard Delorme. edax-reversi. <https://github.com/abulmo/edax-reversi>, 2021. Last retrieved 2023-07-07.
- [23] French Othello Federation. La base wthor. <https://www.ffothello.org/informatique/la-base-wthor/>, 2021. Last retrieved 2023-10-13.