Strongly solved Ostle: calculating a strong solution for composing high-quality puzzles for recent games

Hiroki Takizawa Corresp. [1, 2]

[1] Graduate School of Frontier Sciences, University of Tokyo, Kashiwa city, Chiba prefecture, Japan

[2] Current Affiliation: Preferred Networks, Inc., Chiyoda-ku, Tokyo, Japan

Corresponding Author: Hiroki Takizawa
Email address: contact@hiroki-takizawa.name

Pure strategy board games such as chess are popular intellectual activities, and solving them is a challenging task in computer science. In addition to traditional games, many new board games have gained popularity in recent years. Ostle is one such unsolved game published in 2017. It is based on simple rules but is highly competitive. It is a two-player zero-sum game with perfect information in which the game-theoretical values of all game states can be obtained. In this study, we strongly solved Ostle by retrograde analysis. Utilizing various known techniques, including bitboards and succinct indexable dictionaries, significantly reduced the memory consumption in the analyses. We confirmed that the initial position is a draw and found some fundamental properties of Ostle. Additionally, we composed a tactical Ostle puzzle by using outputs of the analyses. The result demonstrates that solving recent games can be helpful to compose high-quality problems.

Strongly solved Ostle: calculating a strong solution for composing high-quality puzzles for recent games

Hiroki Takizawa[1,2]

[1]**Graduate School of Frontier Sciences, University of Tokyo, Kashiwa city, Chiba prefecture, Japan**
[2]**Current Affiliation: Preferred Networks, Inc., Chiyoda-ku, Tokyo, Japan**

Corresponding author:
Hiroki Takizawa[1,2]

Email address: contact@hiroki-takizawa.name

ABSTRACT

Pure strategy board games such as chess are popular intellectual activities, and solving them is a challenging task in computer science. In addition to traditional games, many new board games have gained popularity in recent years. Ostle is one such unsolved game published in 2017. It is based on simple rules but is highly competitive. It is a two-player zero-sum game with perfect information in which the game-theoretical values of all game states can be obtained. In this study, we strongly solved Ostle by retrograde analysis. Utilizing various known techniques, including bitboards and succinct indexable dictionaries, significantly reduced the memory consumption in the analyses. We confirmed that the initial position is a draw and found some fundamental properties of Ostle. Additionally, we composed a tactical Ostle puzzle by using outputs of the analyses. The result demonstrates that solving recent games can be helpful to compose high-quality problems.

INTRODUCTION

Computational solving of pure strategy board games such as chess, go, and checkers has been one of the goals of computer science. In the early days of computer science, Charles Babbage described the concept of automatically solving board games in his autobiography (Babbage, 1864). Various pure strategy board games are solved to date, and the most famous study is seemingly solving checkers (Schaeffer et al., 2007). In order to solve checkers, they used various algorithms: retrograde analysis (Thompson, 1986), alpha-beta search (Knuth and Moore, 1975) of superhuman-strength checker-program named Chinook (Schaeffer, 1997), and Df-pn (Nagai, 2002; Kishimoto et al., 2012). Df-pn is based on proof-number search (Allis et al., 1994).

Solving games can be categorized as follows (Allis, 1994):

Ultra-weakly solved If the game-theoretic value of the initial position is determined, then the game is ultra-weakly solved. Note that this definition does not require any actual winning strategy.

Weakly solved If a strategy to achieve the game-theoretic value of the game for both players, from the initial position, under reasonable computational resources, then the game is weakly solved. For example, checkers was weakly solved in this sense (Schaeffer et al., 2007).

Strongly solved If the game-theoretic values of all possible legal positions are determined for both players, then the game is strongly solved. Note that a winning strategy can easily be obtained once a strong solution is given (i.e., the theoretical values of the positions after each legal move from the current position can be seen).

Retrograde analysis (Thompson, 1986) is a standard method for strongly solving a pure strategy board game. Notably some studies proposed even stronger categories than the above mentioned ones (Schaeffer and Lake, 1996; Gévay and Danner, 2015). They considered models in which opponent probabilistically makes mistakes, but those models fall outside the scope of this study.

Solving games is a different concept from developing superhuman-strength programs for games, as they do not necessarily require an analytic solution. Therefore, much research reported superhuman-strength programs for large-scale games that seem intractable to solve even ultra-weakly, such as go (Silver et al., 2016), chess (Campbell et al., 2002), and reversi (Buro, 1997). In contrast, developing a superhuman-strength program is straightforward once a game is weakly or strongly solved.

Several popular pure strategy board games have yet to be solved, and the number of such unresolved games continues to increase. In recent years, many new pure strategy board games have emerged, and some have become popular. Ostle, published in Japan in 2017, is one of them. In addition to games created by human game designers, some games generated by AIs have become popular, such as Yavalath (Browne and Maire, 2010; Browne, 2011).

In this study, we strongly solved the game Ostle by retrograde analysis (Thompson, 1986); this is the first work that strongly solved Ostle. We determined that the initial position is a draw. We found positions that take 147 plies to win, assuming that both players always choose the best move, and confirmed that 147 plies are the longest of all positions. Additionally, we performed a breadth-first search and proved that all positions targeted in the retrograde analysis are reachable from the initial position.

In addition, we exhaustively enumerated positions in which sacrifice (i.e., a move that voluntarily loses a piece) is necessary to win (e.g., Figure 1b). The significance of this work is not only that we have discovered interesting positions but also that we obtained helpful information to compose problems involving a tactical factor. For an experimental demonstration, we composed an Ostle problem with this information.

METHODS

The rules of Ostle

Ostle is a two-player game. The players are referred to as Black and White. The initial position of Ostle is shown in Fig. 1a. The five dark gray pieces on the first rank are Black's pieces; the white pieces on the fifth rank are White's. The black circle on c3 is a "hole". Note that ranks(rows) and files(columns) are not labeled with numbers and letters in the official explanation; the labels were added to enable chess-like algebraic notation.

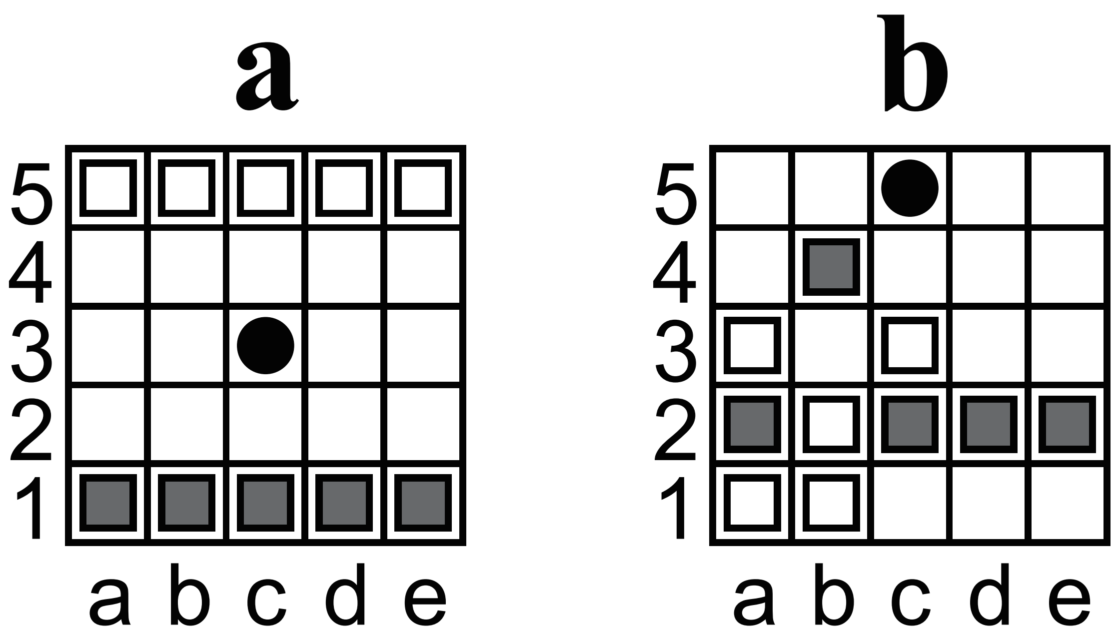

Figure 1. (a) An illustration of the initial position of Ostle. (b) Another example of a position. The position was discovered as a puzzle; if Black is the player to move, there is only one move that leads Black to win in seven plies. The answer is shown in Fig. 6.

The rules of Ostle are as follows:

1. Black moves first, after which the players alternate.
2. On each player's turn, that player must choose either one of his/her owned pieces or the hole to be moved and move it in one square up, down, left, or right.
3. A pass is not allowed; both players must move.
4. If a piece reaches the hole or outside the board, it is removed from the game. Moving a player's piece to the hole or outside the board is allowed.
5. Pieces can be moved to a square occupied by another piece. In this case, the original piece is pushed out and moved one square in the same direction. This process is recursive until a piece reaches an empty square, the hole, or the outside of the board (cf. Figs. 2a and 2c).
6. A player wins when the opponent has only three pieces left. For example, Black wins in Fig. 2d.
7. The holes can only be moved to empty squares. For example, in Fig. 2b, White moves the hole from a3 to a2 but cannot move it to b3 or a4 (because they are not empty) or to the left (because the left is outside the board).
8. Any position must not be the same as two plies before it. Any move that causes such a situation is restricted. For example, in Fig. 2c, Black cannot move the hole to a3 because the position after the ply would be exactly the same as the previous position (Fig. 2b).

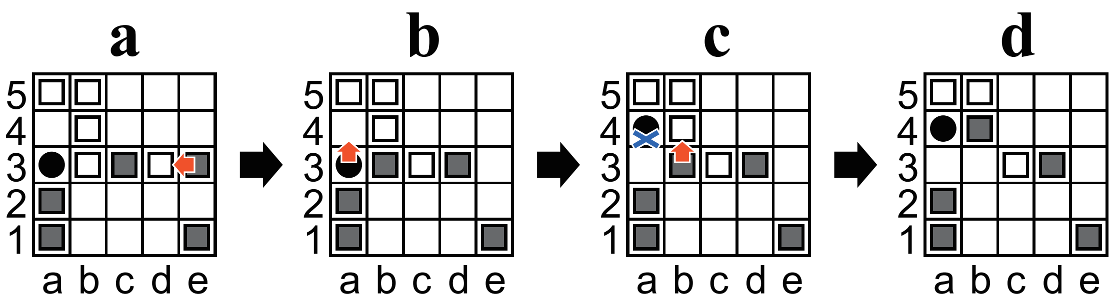

Figure 2. Example diagrams of Ostle. Red arrows represent a chosen move. A blue cross mark represents a restriction.

Rule 8 inhibits one kind of repetition, but some repetitions are still possible. Therefore, Ostle, defined by the above rules, is not a finite game. In the following, we treat every repetition as a draw.

Notation and preliminary

A formal model and useful notation are desribed below. We used the following components:

- A finite set P of **positions**, such that every $p \in P$ corresponds to a unique arrangement of pieces on the board of an unfinished game, including the hole, as well as which player's turn it is. Note that every $p \in P$ includes either four or five white and four or five black pieces.
- A finite set M_p of **moves** associated to every $p \in P$. Moves are represented by an alphanumeric coordinates for the square and an uppercase letter for the direction (U, D, L, or R for up, down, left, or right, respectively). For example, the chosen moves in Fig. 2 are represented as e3L (Fig. 2a), a3U (Fig. 2b), and b3U (Fig. 2c).
- A finite set S of **states**, such that each $s \in S$ consists of a unique tuple (p_s, m_s^r), where $p_s \in P$ and $m_s^r \in (M_{p_s} \cup \{\phi\})$. The first element of the tuple identifies a position, and the second identifies which move is restricted (if applicable).
- A function $f_{transition}$ that takes a position $p \in P$ and its move $m \in M_p$ as arguments, such that $f_{transition}(p, m)$ returns a state $s \in S$ such that p transitions to s by m. A ply is represented by $f_{transition}$.
- A function f_{square} associated with every $p \in P$ takes a move $m \in M_p$ as arguments, such that $f_{square}(m)$ returns m's source square. For example, in Fig. 2a, if m is e3L, then $f_{square}(\text{e3L}) = $e3.
- A function $f_{direction}$ associated with every $p \in P$ that takes a move $m \in M_p$ as argument, such that $f_{direction}(m)$ returns the direction of m. For example, in Fig. 2a, if m is e3L, then $f_{direction}(\text{e3L}) = $L.

- A boolean-valued function f_{if_PRS} takes a state $s \in S$ as an argument, such that $f_{if\_PRS}(s)$ returns True if and only if a position $p \in P$ and an associated move $m \in M_p$ exist such that p transitions to s by m. A **possibly reachable state** is a state s for which $f_{if\_PRS}(s)$ is True. Note that it does not consider whether $(p', m) \in S$ or not. In other words, if s is not a possibly reachable state, s is guaranteed to be unreachable from any state; however, if s is a possibly reachable state, it does not follow that s is guaranteed to be reachable.
- A boolean-valued function g that takes a state $s \in S$ as an argument, such that $g(s)$ returns True if and only if a legal move $m' \in M_{p_s} \setminus \{m^r_s\}$ exists such that m' wins the game for the player to move. A **checkmate state** is a state s for which $g(s)$ is True. For example, Fig. 2c is a checkmate state because Black wins with b3U. Additionally, a position $p \in P$ is called a **checkmate position** if and only if all corresponding states of $\{s \in S : p_s = p \wedge f_{if\_PRS}(s) = True\}$ are checkmate states.
- A boolean-valued function h that takes a state $s \in S$ as an argument, such that $h(s) = f_{if\_PRS}(s) \wedge (\neg g(s))$. A state $s \in S$ is called a **non-trivial state** if and only if $h(s)$ is True.
- Assume that i, j, k are integers and $i \leq j < k$. Here, a bracket notation $[i, j]$ indicates the integer interval between i and j, including both. Another bracket notation $[i, k)$ also indicates an integer interval, but k is excluded. In other words, $[i, k) =$ "$i, i+1, \ldots, k-1$".

Below are several theorems regarding the relationship between a checkmate state and checkmate position.

Theorem 1 *Any move which removes a piece cannot be restricted.*

Proof 1 *In Ostle, the number of pieces on the board decreases monotonically because no ply increases pieces. For this reason, for an arbitrary move m which removes a piece, the position after m is different from the position two plies before m in terms of the number of pieces. Therefore, m is never restricted by rule 8.* *Q.E.D.*

Theorem 2 *Any move which wins the game cannot be restricted.*

Proof 2 *Assume that a move $m \in M_p$ from a position $p \in P$ wins the game. This means that there are just four opponent's pieces in p, and m removes an opponent's piece. By the Theorem 1, we can conclude that m is never restricted.* *Q.E.D.*

Theorem 3 *If a state $s \in S$ is a checkmate state, then the corresponding position p_s is always a checkmate position.*

Proof 3 *The proof is by contradiction. Assume that there exists a state $s' \in S$ such that s' is a checkmate state, but corresponding position $p_{s'}$ is not a checkmate position. Then there must exist a move $m \in M_{p_{s'}}$ such that m wins for the player to move.*

From the definition of a checkmate position, there must exist a state $s^* \in S$ such that $p_{s^*} = p \wedge f_{if\_PRS}(s^*) = True$. Note that $m \in M_{p_{s^*}}$ because a set of moves is associated only with a position, not a state. In order to satisfy the condition $f_{if\_PRS}(s^*) = True$, m must be restricted in the state s^*. This is in contradiction to Theorem 2. *Q.E.D.*

Move generation preliminaries

The details of a move generation algorithm are described below. Algorithm 1 generates all moves of an argument position in a predetermined order.

Note that Algorithm 1 does not consider the restriction of rule 8 and generates a restricted move for computational efficiency. In other words, Algorithm 1 is a "pseudo-legal" move generator; it is guaranteed that Algorithm 1 generates all legal moves, but each generated move is not guaranteed to be legal.

In addition, note that a position $p \in P$ and multiple moves $m_1, \ldots, m_n \in G(p)$ $(2 \leq n \leq 4)$ exist such that for all $i \in [1, n]$, m_i transitions p into the same position $p' \in P$. For example, in the initial position, the three moves "a1D", "a1L", and "a1R" brings the same position (the piece on a1 is removed and everything else remains the same).

However, for an arbitrary position $p \in P$, the number of restricted moves in the return value of $G(p)$ is at most one. In order to show this, there are several theorems in the following.

Algorithm 1 $G(p)$: Generate all moves in a predetermined order.

Require: p: A position. Note that the pieces must be labeled not as Black or White, but as Self (the player to move) or Opponent.

```
a ← an empty list
b ← {'a','b','c','d','e'} × {'1','2','3','4','5'}
b ← sort(list(b))        ▷ A list of all squares' names in lexicographical order.
for s ∈ b do
    if The hole exists on square s then
        if The hole can be legally moved up then
            a.append(s+'U')
        end if
        if The hole can be legally moved down then
            a.append(s+'D')
        end if
        if The hole can be legally moved left then
            a.append(s+'L')
        end if
        if The hole can be legally moved right then
            a.append(s+'R')
        end if
    else if A Self's piece exists on square s then
        for d ∈ {'U','D','L','R'} do
            a.append(s+d)
        end for
    end if
end for
return a
```

Theorem 4 *For all $p \in P$ and $m_1, m_2 \in M_p$, if $f_{square}(m_1) \neq f_{square}(m_2)$, then $f_{transition}(p, m_1) \neq f_{transition}(p, m_2)$.*

Proof 4 *After an arbitrary move, the square from which the piece was moved becomes empty. In contrast, the other squares never become empty if they were originally not empty. Therefore, $f_{transition}(p, m_1) \neq f_{transition}(p, m_2)$ in terms of whether the square from which the piece was moved is empty. Q.E.D.*

Theorem 5 *For all $p \in P$ and $m_1, m_2 \in M_p$ ($m_1 \neq m_2$), if neither m_1 nor m_2 remove any pieces, then $f_{transition}(p, m_1) \neq f_{transition}(p, m_2)$.*

Proof 5 *According to theorem 4, if $f_{square}(m_1) \neq f_{square}(m_2)$, then $f_{transition}(p, m_1) \neq f_{transition}(p, m_2)$. In the following, we will consider the case where $f_{square}(m_1) = f_{square}(m_2)$. Let us denote $Q = f_{square}(m_1) = f_{square}(m_2)$. Because $m_1 \neq m_2$, m_1 and m_2 differ in the directions. Because neither m_1 nor m_2 remove any pieces, for each $i \in \{1, 2\}$, there is one square that is empty at p but filled at $f_{transition}(p, m_i)$; let us denote the square by Q_i. Note that Q_i is in the direction of m_i from Q. Then $Q_1 \neq Q_2$ if $m_1 \neq m_2$, and m_1 and m_2 are in different directions. Consequently, if $m_1 \neq m_2$ and $f_{square}(m_1) = f_{square}(m_2)$, then $f_{transition}(p, m_1) \neq f_{transition}(p, m_2)$ in terms of the square that is empty at p but filled at $f_{transition}(p, m_i)$. Q.E.D.*

Theorem 6 *For all $p \in P$, the number of restricted moves in $G(p)$ is at most one.*

Proof 6 *Taking the contraposition of theorem 5, we can find that for all $m_1, m_2 \in G(p)$ ($m_1 \neq m_2$)), if $f_{transition}(p, m_1) = f_{transition}(p, m_2)$, then m_1 and m_2 are moves removing a piece. Using theorem 1, we can find that such m_1 and m_2 are never restricted. Taking the contraposition of this, we can conclude that for all $m, m' \in G(p)$ ($m \neq m'$), if m is restricted, then $f_{transition}(p, m) \neq f_{transition}(p, m')$, hence $f_{transition}(p, m')$ is not restricted. Therefore, for all $p \in P$ and $m \in G(p)$, if m is restricted, then all the other moves in $G(p)$ never restricted. Q.E.D.*

The return value of Algorithm 1 is a list of moves. It is essential for further analysis that the order, as well as the members, is deterministic, because in further analysis, every state in an arbitrary position is assigned a unique serial number based on the index of the restricted move in the list. This numbering method works correctly only if at most one move is restricted. Although Algorithm 1 is a pseudo-legal move generator, the number of restricted moves is guaranteed to be at most one by Theorem 6.

Positional Symmetry

Symmetry inherent in Ostle can make further analysis, including retrograde analysis, more efficient without losing any essential information. Specifically, each position has at most eight symmetric positions, including itself. Algorithm 2 enumerates these positions.

Algorithm 2 $E(p)$: Enumerate all symmetric positions.

Require: p: A position (e.g., a 5×5 matrix).
Require: $flip\_lr(p)$: A function that horizontally flips the argument position.
Require: $flip\_ud(p)$: A function that vertically flips the argument position.
Require: $transpose(p)$: A function that transposes the argument position.

$a \leftarrow$ an empty set
for $i \leftarrow [0,7]$ **do**
  $x \leftarrow p$
  if $(i \& 1) \neq 0$ **then** ▷ The "&" symbols refer to the bitwise-and operation.
    $x \leftarrow flip\_lr(x)$
  end if
  if $(i \& 2) \neq 0$ **then**
    $x \leftarrow flip\_ud(x)$
  end if
  if $(i \& 4) \neq 0$ **then**
    $x \leftarrow transpose(x)$
  end if
  a.add(x)
end for
return a

In the following analysis, it was often helpful to consider symmetric positions as identical. Algorithm 3 was used to obtain a unique representative position among the symmetric positions.

Algorithm 3 $U(p)$: Get a unique representative position among the symmetric positions.

Require: p: A position (e.g., a 5×5 matrix).
Require: $ptoi(p)$: An injective function that maps positions to integers.

$a \leftarrow p$
for $x \in E(p)$ **do**
  if $ptoi(x) < ptoi(a)$ **then**
    $a \leftarrow x$
  end if
end for
return a

Algorithm 3 calls a "ptoi" function that injectively maps positions into integers. Any mapping is acceptable as long as it is injective. In our implementation for this study, we represented a position itself to be a 55-bit integer. Two bitboards of pieces needed 25 bits each, and the remaining five bits were for the square of the hole. Therefore, the "ptoi" function was not employed (in other words, it was an identity map).

Enumerating positions

Before the analysis, the possible positions of Ostle were exhaustively enumerated. Enumeration was based on the following criteria.

- Checkmate positions were included in the enumeration. In contrast, positions after the game were over (i.e., positions where a loser had only three pieces) were not included.
- Both players had no obligation to win in any checkmate position. In other words, positions that are unreachable from the initial position without overlooking a winning move were included in the enumeration.
- Only positions in which it was Black's turn to move were enumerated. In other words, in the following, "Black" means "the player to move", and "White" means "their opponent", except where specifically noted otherwise. This is sufficient because there is a sequence of moves whereby the same position is reached, but the player to move is changed (an example is shown in Fig. 3).
- Symmetric positions were considered identical. This is sufficient because there is a sequence of moves to rotate the initial position ninety degrees (an example is shown in Fig. 4). The other symmetric positions can be reached by repeating the sequence two or three times.

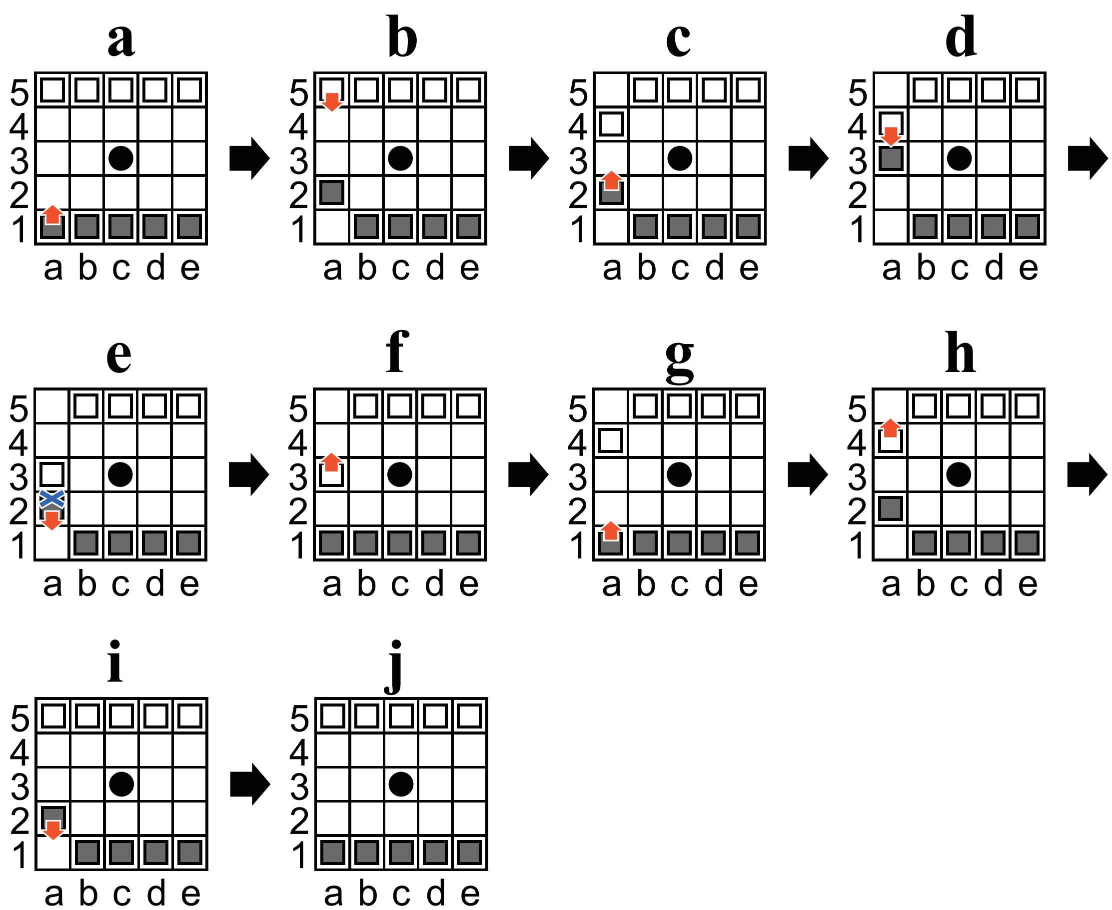

Figure 3. Diagrams of Ostle illustrating a sequence to change the player to move in nine plies. Figures 3a and 3j are identical, except for the player to move; figure 3a is Black's turn, but Fig. 3j is White's turn.

The two observations below enable us to reduce the number of enumerating positions without losing exhaustiveness.

Observation 1

Let us consider the procedure of choosing the place of the hole first and then the pieces (white and black, five or four pieces each) in the remaining 24 squares. This procedure can construct an arbitrary position, but symmetric positions are enumerated separately. Here, if the symmetric positions are to be considered identical later, only six squares, a1, a2, a3, b2, b3, and c3, are sufficient to be considered for the hole placement. This is because if the hole is placed on one of the remaining 19 squares and the pieces on arbitrary squares, there always exist a symmetric position such that the hole is on one of the six squares.

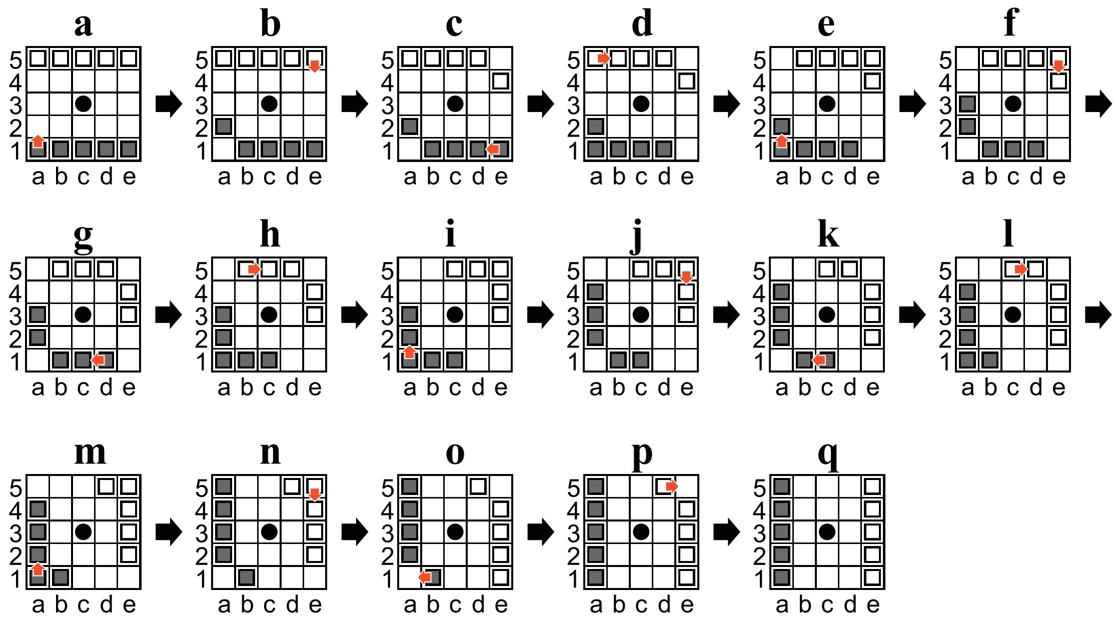

Figure 4. Diagrams of Ostle illustrating a sequence to rotate the initial position ninety degrees in sixteen plies.

Observation 2

There is never a symmetric relationship between two boards if their holes are on different squares of the above six squares. Moreover, there is never a symmetric relationship between two boards if the number of pieces of at least one player is different. Therefore, the positions could be divided into 24 cases according to the “place of the hole and number of pieces”. This categorization is mutually exclusive and collectively exhaustive. The detection of symmetric positions could be done by considering only the inside of each divided subset.

Algorithms

Algorithm 4 exhaustively enumerates and sorts all possible positions. In the description, each algebraic coordinate was assigned a number in lexicographical order; a1 was assigned 0, a2 was assigned 1, b1 was assigned 5, and so on.

The reason for sorting the list of positions was to perform a binary search to find the index of an arbitrary position. Therefore, any sorting criterion is acceptable as long as the comparison is fast.

Algorithm 4 Enumerate and sort all positions.

```
v ← an empty list
for (b,w,h) ∈ {4,5} × {4,5} × {0,1,2,6,7,12} do          ▷ This for-loop is parallelizable.
    w ← an empty list
    p ← an empty position (e.g., a dictionary)
    w ← D(w, p, 0, b, w|h)                                  ▷ The function D is Algorithm 5.
    for i ∈ [0, len(w)) do
        w[i] ← U(w[i])                                      ▷ The function U is Algorithm 3.
    end for
    w.uniquify()    ▷ e.g., in C++, std::sort and std::unique are available; in Python, list(set(w)) is.
    v.concatenate(w)               ▷ If executed in parallel, this line must be in a critical section.
end for
return sort(v)
```

Algorithm 5 $D(v, p, n, b, w|h)$: An auxiliary function of depth-first search to enumerate positions.

Require: v: A list of positions.
Require: p: An in-process position (e.g., a dictionary).
Require: n: An integer representing a considering square ($n \in [0, 25]$).
Require: b: An integer representing a number of remaining Black's pieces ($b \in [0, 5]$).
Require: w: An integer representing a number of remaining White's pieces ($w \in [0, 5]$).
Require: h: An integer representing the square where the hole exists ($h \in \{0, 1, 2, 6, 7, 12\}$).

if $n = 25$ **then**
    v.append(p)
    return v
end if
if $n = h$ **then**
    $p[n] \leftarrow$ "*hole*"
    return $D(v, p, n+1, b, w|h)$
end if
if $b > 0$ **then**
    $q \leftarrow p$
    $q[n] \leftarrow$ "*black*"
    v.concatenate($D(v, q, n+1, b-1, w|h)$)
end if
if $w > 0$ **then**
    $q \leftarrow p$
    $q[n] \leftarrow$ "*white*"
    v.concatenate($D(v, q, n+1, b, w-1|h)$)
end if
v.concatenate($D(v, p, n+1, b, w|h)$)
return v

Enumerating non-trivial states

2,735,147,685 positions were enumerated by Algorithm 4 (discussed in more detail in the Result section). From Theorem 6, we can say that each position $p \in P$ contains at most 25 states; one of them has no restricted move, and the others have one restricted move. Note that the number of moves generated by Algorithm 1 is at most 24. Therefore, the number of states is at most 68,378,692,125 ($= 2,735,147,685 \times 25$).

In the following, we define a one-to-one correspondence between states and $[0, |P| \times 25)$. Specifically, if a state s has no restriction move ($s = (p_s, \phi)$), s is mapped to $j \times 25$, where $P[j] = p_s$. Otherwise ($s = (p_s, m_s^r)$), s is mapped to $j \times 25 + k + 1$, where $P[j] = p_s$, $k \in [0, |G(p_s)|)$, and $G(p_s)[k] = m_s^r$. In the following, we denote a state s as the i-th state if and only if s is mapped to i.

We hypothesized that the number of non-trivial states is significantly smaller than 68,378,692,125. In order to confirm this, we developed Algorithm 6, which takes a sorted list of positions as an argument and returns a bitvector that represents whether each state is non-trivial. Note that the outermost for-loop of Algorithm 6 is parallelizable, but if parallelized, the operation of setting a bit of v must be atomic or executed in a critical section.

Bitvector and succinct indexable dictionary

Algorithm 6 returns a bitvector v, whose length is 68,378,692,125 (It equals $|P| \times 25$). Let us denote the i-th bit of v as $v[i]$. For all $i \in [0, |v|)$, it is guaranteed that $v[i] = 1$ if and only if the i-th state is non-trivial. Consequently, 11,148,725,918 states were non-trivial (discussed in more detail in the Result section).

For retrograde analysis, an array must be allocated to record the theoretical values of game states. If a 16-bit integer is allocated for each state, it will consume more than 136 GB of RAM. However, if allocated only for non-trivial states, memory consumption would be reduced to less than 23 GB of RAM. Let us denote the array only for non-trivial states as x.

To access the i-th state on v, it is necessary to find its index number on x. The index equals the number of bits standing in the range $[0, i)$ on v. It can be obtained by a query called a "rank query to a bitvector",

Algorithm 6 $B(P, c = \text{False})$: Make a bitvector that represents whether each state is non-trivial.

Require: P: A sorted list of all positions (return value of Algorithm 4).
Require: c: A boolean flag to control whether checkmate positions are counted.

$v \leftarrow$ a zero-filled bitvector whose length is $|P| \times 25$.
for $p \in P$ **do** ▷ This for-loop is parallelizable; but if parallelized, the operation of setting a bit of v must be atomic or executed in a critical section.
  $M \leftarrow G(p)$ ▷ Generate all moves of the position p.
  for $m \in M$ **do**
    $s \leftarrow f_{transition}(p, m)$
    if $c =$True or $g(s) =$False **then**
      $i \leftarrow$ the integer such that $P[i] = U(E(p_s))$ ▷ e.g., perform a binary search.
      $M' \leftarrow G(P[i])$ ▷ Generate all moves of the position $P[i]$. Note that $|M'| \leq 24$.
      for $j \in [0, |M'|)$ **do**
        $s' \leftarrow f_{transition}(P[i], M'[j])$
        if $U(E(p_{s'})) = p$ **then**
          $v[i \times 25 + j + 1] \leftarrow 1$ ▷ Set the $(i \times 25 + j + 1)$-th bit of v to 1.
          goto END:
        end if
      end for
      $v[i \times 25] \leftarrow 1$ ▷ Set the $(i \times 25)$-th bit of v to 1.
      END:
    end if
  end for
end for
return v

which returns the number of bits standing from the top to the i-th bit in the bitvector.

Under the assumption that the bitvector is unchanged after initialization, it is known that the rank query can be processed in constant time by providing an auxiliary data structure called "succinct indexable dictionary" (Jacobson, 1988). It is also known to have the advantage that the space complexity of "succinct indexable dictionary" can be reduced to $o(|v|)$. Nevertheless, since we only consider solving Ostle in the present study, it is unnecessary to focus on computational complexity theory (as it is the asymptotic behavior when $|v|$ goes to infinity). In our implementation, the additional size of the dictionary for v was $\frac{257}{1024}$ of the size of v itself.

Let us denote a bitvector with its succinct indexable dictionary as a "succinct bitvector".

Retrograde analysis

For all non-trivial states, retrograde analysis was performed to obtain the theoretical value of the state. We determined whether each state was a win, a loss, or a draw for the player to move. We also determined the number of plies required to reach a checkmate state, assuming that the winner is minimizing and the loser is maximizing it.

In a naive implementation of retrograde analysis, a game graph (i.e., a directed graph with the states as nodes and the moves as edges) is initially constructed. However, in this study, we implemented Algorithm 7, which performs retrograde analysis of Ostle without explicitly having a game graph.

Algorithm 7 returns a vector of integers; each integer represents the theoretical value of a corresponding state. If it is zero, the state is a draw. If it is a negative number, the state is a loss for the player to move. If it is a positive number, the state is a win for the player to move.

Breadth-first search to prove the reachability

In order to prove that all possibly reachable states are reachable from the initial state, a breadth-first search was performed. Algorithm 7 represents the breadth-first search. The initial position is assumed to be the starting point (distance is zero), and the distance is assumed to be increased by one for each transition. We calculated the minimum distance of all possibly reachable states. Consequently, it was confirmed that all possibly reachable states are reachable from the initial position by finite plies.

Algorithm 7 Retrograde analysis of Ostle

Require: $A(i, v, x, y, l_{win}, l_{lose}, l_{draw})$: Auxiliary function, which is described below as Algorithm 8.

$P \leftarrow$ A sorted list of all positions (i.e., return value of Algorithm 4).
$v \leftarrow B(P)$ ▷ $B(P)$ is Algorithm 6.
Convert v into succinct bitvector which supports **rank** query in constant time.
$x \leftarrow$ A zero-filled vector of which length is $popcount(v) = 11,148,725,918$.
while True do
    $y \leftarrow x$
    for $i \in [0, |P|)$ **do** ▷ This for-loop is parallelizable.
        $l_{win}, l_{lose}, l_{draw} \leftarrow$ empty lists.
        $M \leftarrow G(P[i])$ ▷ generate all moves of the position $P[i]$.
        for $j \in [0, |M|)$ **do**
            $s \leftarrow f_{transition}(P[i], M[j])$
            $k \leftarrow$ the integer such that $P[k] = U(E(p_s))$ ▷ perform a binary search.
            if $g(s)$ **then** ▷ s is a checkmate state.
                l_{lose}.append($(-1, j)$)
            else
                if $r \in [0, 24)$ exists such that $G(P[k])[r]$ is a restricted move **then**
                    $k \leftarrow k + r + 1$
                end if
                if $x[v.rank(k)]$ is a negative number **then**
                    l_{win}.append($(-x[v.rank(k)] + 1, j)$)
                else if $x[v.rank(k)]$ is a positive number **then**
                    l_{lose}.append($(-x[v.rank(k)] - 1, j)$)
                else
                    l_{draw}.append((j))
                end if
            end if
        end for
        Sort the elements of l_{win} and ones of l_{lose} in ascending order.
        $y \leftarrow A(i, v, x, y, l_{win}, l_{lose}, l_{draw})$ ▷ If parallelized, this line must be in a critical section.
    end for
    if $y = x$ **then** ▷ i.e., no state was updated in this iteration.
        break
    end if
    $x \leftarrow y$
end while
return x

Computational resource

A c5.9xlarge instance of Amazon EC2 was used for all analyses performed in this study. The specifications were Intel Xeon Platinum 8124M CPU @ 3.00GHz, 18 physical cores, two threads per core, and 72 GB of RAM.

We also executed the same analyses on a PC, of which specifications were AMD Ryzen 5950X CPU @ 3.40GHz, 16 physical cores, two threads per core, and 128 GB of RAM. We verified that the results were the same as the ones obtained by the former analyses.

RESULTS

Enumerating positions

We firstly enumerated all possible positions by Algorithm 4. Table 1 shows the result. In total, 2,735,147,685 positions were obtained. In those enumerated positions, 399,102,582 were checkmate positions (14.5916 %). Note that the enumerated positions included ones that were unreachable from the initial position (For example, the position in Figure 5a is unreachable if Black is the player to move).

Algorithm 8 $A(i, v, x, y, l_{win}, l_{lose}, l_{draw})$: Auxiliary function for retrograde analysis of Ostle

Require: $i, v, x, y, l_{win}, l_{lose}, l_{draw}$: variables appear in Algorithm 7.
Require: Elements of l_{win} and ones of l_{lose} is already sorted in ascending order.

$\lambda(n)$ = if $n \leq 0$, return inf; otherwise, return n. ▷ A function used in the following.
$\lambda_{ternary}(a, b, c)$ = if a is True, return b; otherwise, return c. ▷ A function used in the following.
for $j \in [0, 25)$ **do**
  if $v[i \times 25 + j] = 1$ **then** ▷ Only if the best move is restricted, choose the second-best move.
    $k \leftarrow v.rank(i \times 25 + j)$
    if l_{win} has two or more elements **then**
      $y[k] \leftarrow \lambda_{ternary}(j = 1 + l_{win}[0][1], \min(\lambda(x[k]), l_{win}[1][0]), \min(\lambda(x[k]), l_{win}[0][0]))$
    else if l_{win} has only one element, and l_{draw} has one or more elements **then**
      $y[k] \leftarrow \lambda_{ternary}(j = 1 + l_{win}[0][1], \max(x[k], 0), \min(\lambda(x[k]), l_{win}[0][0]))$
    else if l_{win} has only one element, and l_{draw} has no element **then**
      $y[k] \leftarrow \lambda_{ternary}(j = 1 + l_{win}[0][1], \max(x[k], l_{lose}[0][0]), \min(\lambda(x[k]), l_{win}[0][0]))$
    else if l_{win} has no element, and l_{draw} has two or more elements **then**
      $y[k] \leftarrow \max(x[k], 0)$
    else if l_{win} has no element, and l_{draw} has only one element **then**
      $y[k] \leftarrow \lambda_{ternary}(j = 1 + l_{draw}[0], \max(x[k], l_{lose}[0][0]), \max(x[k], 0))$
    else if Neither l_{win} nor l_{draw} has any element **then**
      $y[k] \leftarrow \lambda_{ternary}(j = 1 + l_{lose}[0][1], \max(x[k], l_{lose}[1][0]), \max(x[k], l_{lose}[0][0])$
    else
      assert False
    end if
  end if
end for
return y

Obtaining non-trivial states

Because each position contains at most 25 states, the total number of states is at most 68,378,692,125 ($= 2,735,147,685 \times 25$). However, the total number of non-trivial states could be much smaller. For the above reasons, we examined the total number of non-trivial states with Algorithm 6. Consequently, we confirmed that the number of non-trivial states was 11,148,725,918 ($\approx 0.163 \times 68,378,692,125$). As explained in the Method section, we reduced the memory usage of the following analysis by using this fact and the succinct indexable dictionary (Jacobson, 1988).

Retrograde analysis

A retrograde analysis was performed, shown in Algorithm 7, to determine the theoretical values of all game states. The initial position is a draw. Table 2 shows the result. The summation of numbers of states in table 2 equals the number of non-trivial states. Note that the "number of plies" in table 2 is the number of plies to reach a checkmate state, so positions of an even number of plies are winning and those of an

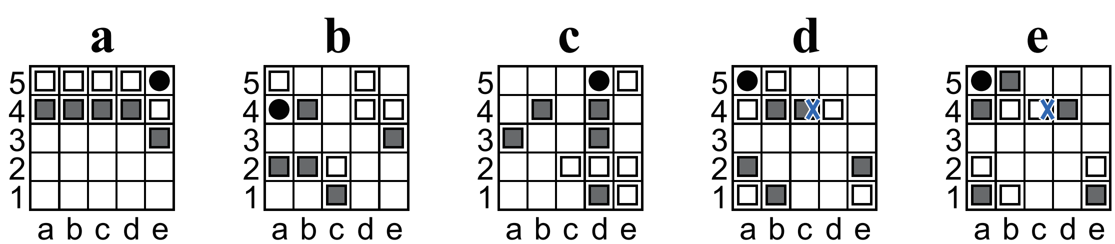

Figure 5. Diagrams of Ostle illustrating special positions and states. (a) An unreachable position if Black is the player to move. (b,c) States taking 147 plies to win (Black is the player to move). (d,e) States taking numerous plies to reach from the initial position (Black is the player to move in Fig. 5d, and White is the player to move in Fig. 5e).

Algorithm 9 Breadth-first search of Ostle

```
P ← A sorted list of all positions (i.e., return value of Algorithm 4).
v ← B(P, True)                                        ▷ B(P) is Algorithm 6.
Convert v into succinct bitvector which supports rank query in constant time.
x ← A inf-filled vector of which length is popcount(v).
I ← The index such that P[I] is the initial position shown in Fig.1a.
x[v.rank(I × 25)] ← 0
for d ∈ [0, inf) do
    y ← x
    for i ∈ [0, |P|) do                               ▷ This for-loop is parallelizable.
        M ← G(P[i])                                   ▷ generate all moves of the position P[i].
        for j ∈ [0, 25) do
            if x[v.rank(i × 25 + j)] = d then
                M′ ← G(P[i])                          ▷ generate all moves of the position P[i].
                for k ∈ [0, |M′|) \ {j − 1} do
                    s ← f_transition(P[i], M[k])
                    l ← the integer such that P[l] = U(E(p_s))   ▷ e.g., perform a binary search.
                    a ← v.rank(l × 25 + k)
                    y[a] ← min(d + 1, x[a])           ▷ If parallelized, this line must be in a critical section.
                end for
            end if
        end for
    end for
    if y = x then                                     ▷ i.e., no state was updated in this iteration.
        break
    end if
    x ← y
end for
return x
```

odd number are losing. The analysis took 35 h to compute using the computing environments described above. The file output of the result was 97.4 GB in size.

States taking 147 plies to win

Table 2 shows that there are seven states taking 147 plies to win, and 147 is the longest number. The seven states comprises two positions shown in Figs. 5b and 5c. The two positions with no restricted move are included in the seven states. The remaining five states consist of the two positions with some restricted move.

Breadth-first search

In order to prove that all possibly reachable states obtained are reachable from the initial position, a breadth-first search was performed (shown in Algorithm 9), which confirmed that all of them were reachable. Table 3 shows the results of the breadth-first search. The analysis took at most 6 h to compute using the computing environments described above. The file output of the result was 114 GB in size.

Table 3 shows that there are 97 states that take 26 plies to reach from the initial position, and 26 is the largest number. For example, the state shown in Fig. 5d is one of the 97 states. Note that the algorithm identifies symmetric positions and which player is the player to move. Therefore, if they were not identified, a state would be found that takes 27 or more plies to reach from the initial position. However, all states identified by this algorithm are guaranteed to be reachable; by adding the sequences shown in Fig. 3 and 4 in the opening, we can say that all symmetric states (and states in which the player to move is changed) are also reachable. The above matters are explained in more detail below.

Algorithm 9 outputs that the state shown in Fig. 5d takes 26 plies to reach from the initial position. In the eight symmetric states, including the one shown in Fig. 5d, there exists a state that can be reached in 26 moves from the initial position, and 26 moves is the least for the eight positions. In other words,

Table 1. The number of enumerated positions.
C* means coordinate of the hole. P* is the number of pieces of the player to move. O* means ones of the opponent. N* is the number of enumerated positions.

C*	P*	O*	N*
a1	5	5	247,127,256
a2	5	5	494,236,512
a3	5	5	247,127,256
b2	5	5	247,127,256
b3	5	5	247,127,256
c3	5	5	61,788,564
a1	5	4	82,378,152
a2	5	4	164,745,504
a3	5	4	82,378,152
b2	5	4	82,378,152
b3	5	4	82,378,152
c3	5	4	20,598,588
a1	4	5	82,378,152
a2	4	5	164,745,504
a3	4	5	82,378,152
b2	4	5	82,378,152
b3	4	5	82,378,152
c3	4	5	20,598,588
a1	4	4	25,744,590
a2	4	4	51,482,970
a3	4	4	25,744,590
b2	4	4	25,744,590
b3	4	4	25,744,590
c3	4	4	6,438,855

P*	O*	N*
5	5	1,544,534,100
5	4	514,856,700
4	5	514,856,700
4	4	160,900,185

all the symmetric states are reachable in just 26 moves from the initial position or one of its symmetric positions. Figure 4 implies that all four positions are reachable. Consequently, we can say that all the eight symmetric states are reachable.

The state shown in Fig. 5e (where White is the player to move) was also treated as the same state as the one in Fig. 5d. By adding the sequence of moves shown in Fig. 3 in the opening, we can say that the state shown in Fig. 5e (where White is the player to move) is reachable from the initial position.

As a side note, every state in which White is the player to move takes an odd number of plies to reach from the initial position. Because the result shows that 26 plies is the least number needed to reach the identified states, The state shown in Fig. 5e (White is the player to move) must take at least 27 plies.

Composing a tactical problem

To demonstrate the usefulness of the retrograde analysis, we discovered interesting states from which to explore the nature of games. For example, we discovered states that are wins for the player to move, but he/she must choose sacrificing a piece to win. We discovered 35,107 such states, and Fig. 1b is an example. Figure 1b can be interpreted as a composed problem with a suitable stipulation ("Win in seven plies.").

Solution to the puzzle in Figure 1b

Solution to the puzzle in Figure 1b is "d2L,a1R,c2L,b1R,b2R,c1U,c3U" as shown in Fig. 6. In the solution, White chose the moves that maximize the number of plies before losing, and Black chose ones that minimize the number of plies. At the first position (Fig. 6a), “d2L” and “e2L” are the only choices to win. “e2L” is also a sacrificing move but takes nine plies to win.

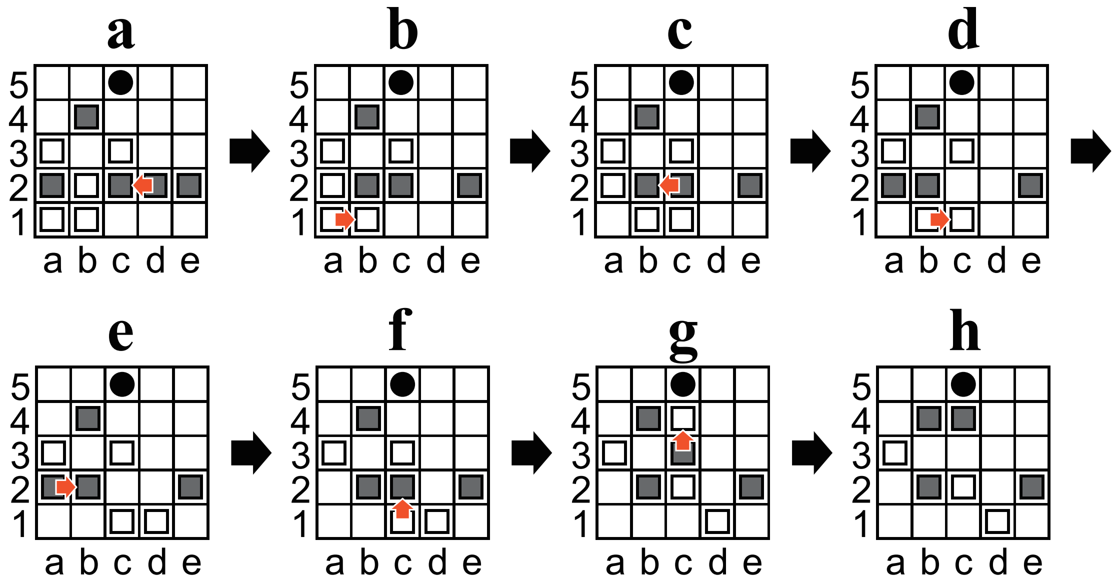

Figure 6. Diagrams of Ostle illustrating a sequence in which that Black wins in seven plies. Figure 6a is the same position as Fig. 1b.

DISCUSSION AND FUTURE WORKS

This study is the first to strongly solve Ostle and determine the theoretical values of all game states. We discovered that the initial position is a draw. Additionally, we found that there exist states from which it takes 147 plies to win, and those that take at least 26 plies to reach from the initial position.

Based on the results of the performed analysis, we discovered states in which sacrificing is necessary to win. Because sacrificing is a highly tactical move, we can say that the significance of the discovery is rooted in the nature of games. There might exist other tactics apart from sacrificing and seeking them is one of the goal for future work.

To demonstrate the usefulness of strongly-solving games, we composed a tactical puzzle of Ostle, which consisted of a state discovered through the analysis and a suitable stipulation. We want to emphasize that strongly solving is a promising tool for composing puzzles, especially for recent popular pure strategy board games.

Based on the output of our retrograde analysis, software can easily be made that instantly chooses the best move for an arbitrary state of Ostle. Because the initial position is a draw, such software never loses. However, it can’t win unless the opponent makes a mistake. Here, it will be a future challenge to create a program that can lure the human opponent into a state where he/she is likely to make a mistake.

We solved Ostle in this study, but in principle every pure strategy board game is solvable. The reason we cannot solve larger games (i.e., reversi, chess, and go) is due to lack of computing power and algorithms. Since computing power continues to improve year by year, solving larger famous games using future computer must be grand challenges achieved in the future.

CONCLUSIONS

Many pure strategy board games are still unsolved and have interesting but undiscovered aspects, but computing power is limited. Therefore, to solve a wider variety of games at a more detailed level, it is essential to use techniques to reduce memory consumption and computation time for each game. In this study, we considered various properties of the subject game (such as the symmetry of the positions) and utilized various techniques such as succinct data structure and bitboards. Consequently, the analysis of Ostle could be performed in an inexpensive computing environment. We hope that this paper and the source code help with future research on solving, analyzing, and extracting interesting facts from various other pure strategy board games.

ACKNOWLEDGEMENTS

The author gratefully thanks Prof. Kiyoshi Asai and Dr. Toshiyuki T. Yokoyama for useful discussion.

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 is available at GitHub: https://github.com/eukaryo/ostle_solver . The outputs of analyses are available at figshare: https://doi.org/10.6084/m9.figshare.19668789.v1 .

Funding

The author has received no funding for this study.

REFERENCES

Allis, L., van der Meulen, M., and van den Herik, H. (1994). Proof-number search. *Artificial Intelligence*, 66(1):91–124.

Allis, L. V. (1994). *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, Department of Computer Science, University of Limburg.

Babbage, C. (1864). *Passages from the Life of a Philosopher*. London: Longman.

Browne, C. (2011). *Evolutionary game design*. Springer London.

Browne, C. and Maire, F. (2010). Evolutionary game design. *Computational Intelligence and AI in Games, IEEE Transactions on*, 2:1 – 16.

Buro, M. (1997). The othello match of the year: Takeshi murakami vs. logistello. *ICGA Journal*, 20(3):189–193.

Campbell, M., Hoane, A. J., and Hsu, F.-h. (2002). Deep blue. *Artif. Intell.*, 134(1–2):57–83.

Gévay, G. E. and Danner, G. (2015). Calculating ultrastrong and extended solutions for nine men's morris, morabaraba, and lasker morris. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(3):256–267.

Jacobson, G. J. (1988). *Succinct static data structures*. PhD thesis, Carnegie Mellon University.

Kishimoto, A., Winands, M. H., Müller, M., and Saito, J.-T. (2012). Game-tree search using proof numbers: The first twenty years. *Icga Journal*, 35(3):131–156.

Knuth, D. E. and Moore, R. W. (1975). An analysis of alpha-beta pruning. *Artificial intelligence*, 6(4):293–326.

Nagai, A. (2002). *Df-pn algorithm for searching AND/OR trees and its applications*. PhD thesis, Department of Information Science, University of Tokyo.

Schaeffer, J. (1997). One jump ahead: Challenging human supremacy in checkers. *ICGA Journal*, 20(2):93–93.

Schaeffer, J., Burch, N., Björnsson, Y., Kishimoto, A., Müller, M., Lake, R., Lu, P., and Sutphen, S. (2007). Checkers is solved. *science*, 317(5844):1518–1522.

Schaeffer, J. and Lake, R. (1996). Solving the game of checkers. *Games of no chance*, 29:119–133.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489.

Thompson, K. (1986). Retrograde analysis of certain endgames. *J. Int. Comput. Games Assoc.*, 9(3):131–139.

Table 2. **Number of plies to reach any of the checkmate positions, and the number of such states.** inf means draw. P* is the number of plies. N* is the number of such states.

P*	N*	P*	N*	P*	N*	P*	N*
inf	339,367,091	40	31,679,932	80	352,810	120	690
1	577,327,477	41	27,408,599	81	291,770	121	425
2	1,208,259,074	42	25,272,339	82	285,312	122	433
3	250,385,204	43	21,858,443	83	233,005	123	282
4	514,915,495	44	20,072,851	84	229,210	124	387
5	294,380,826	45	17,341,986	85	186,949	125	308
6	569,040,388	46	15,898,815	86	179,324	126	321
7	352,821,271	47	13,731,415	87	144,595	127	212
8	559,455,180	48	12,600,718	88	138,428	128	226
9	379,808,723	49	10,908,242	89	113,388	129	205
10	535,425,081	50	9,994,880	90	108,165	130	179
11	379,563,356	51	8,667,693	91	86,970	131	211
12	462,977,806	52	7,940,165	92	83,620	132	73
13	352,330,833	53	6,893,605	93	67,110	133	113
14	395,732,654	54	6,314,317	94	65,247	134	44
15	318,063,321	55	5,502,897	95	53,164	135	99
16	338,626,104	56	5,039,556	96	51,284	136	68
17	281,546,559	57	4,382,022	97	40,837	137	146
18	287,320,623	58	3,998,228	98	38,653	138	74
19	244,575,415	59	3,470,949	99	30,388	139	106
20	241,883,809	60	3,163,370	100	27,311	140	44
21	208,979,928	61	2,764,844	101	22,936	141	40
22	202,402,295	62	2,521,503	102	21,612	142	16
23	176,160,115	63	2,195,886	103	18,273	143	7
24	168,414,137	64	2,010,460	104	15,740	144	9
25	146,956,825	65	1,759,413	105	13,176	145	4
26	139,319,551	66	1,595,483	106	10,996	146	7
27	121,669,807	67	1,400,525	107	9,486		
28	114,485,394	68	1,267,104	108	7,740		
29	99,874,325	69	1,108,248	109	7,290		
30	93,577,296	70	1,006,232	110	5,481		
31	81,527,252	71	878,861	111	5,015		
32	76,116,163	72	801,288	112	3,885		
33	66,122,478	73	700,241	113	3,741		
34	61,530,140	74	649,879	114	2,885		
35	53,329,309	75	561,353	115	2,593		
36	49,530,287	76	528,175	116	1,785		
37	42,847,011	77	449,395	117	1,447		
38	39,660,828	78	434,621	118	1,005		
39	34,312,611	79	361,217	119	771		

Table 3. **Number of plies to reach states from the initial state, and the number of such states.** P* is the number of plies. N* is the number of such states.

P*	N*	P*	N*
0	1	14	411,886,389
1	9	15	767,525,717
2	102	16	1,262,744,615
3	954	17	1,851,900,832
4	6,329	18	2,259,589,185
5	33,052	19	2,356,709,939
6	147,620	20	1,884,609,912
7	556,811	21	1,172,437,043
8	1,863,530	22	475,193,903
9	5,542,830	23	113,051,575
10	15,200,179	24	9,503,831
11	38,307,337	25	115,519
12	91,419,758	26	97
13	201,637,267		