



Artificial Intelligence  
(CSC-4753)

# **Othello Done Fast**

**Squeezing out maximum performance  
using better datastructures**

31st December 1979

Course: Artificial Intelligence  
Author: Jack Branch

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	A Naive Solution . . . . .	1
1.2	The Optimal Solution . . . . .	1
1.3	A Note on Notation . . . . .	2
<b>2</b>	<b>Board Operations</b>	<b>3</b>
2.1	Generating Moves . . . . .	3
2.1.1	The Optimal Solution . . . . .	4
2.1.2	Implementation in Rust . . . . .	7
2.2	Playing Moves . . . . .	8
2.2.1	Benchmarks . . . . .	8

Some of the datastructures and algorithms in use for this assignment are a bit unintuitive, so I figured I would spend the time thoroughly explaining each.

## 1 Introduction

In standard Othello, you have an eight by eight board of squares. Each square can contain a disc that is either black or white. How might you represent this structure in memory? This question and the implications of its answers is what this document will explore.

I will make references to Rust structures and types while using pseudocode for all algorithms.

### 1.1 A Naive Solution

A common naive approach would be to represent it as an array of characters where a `b` character represents a black disc, a `w` character represents a white disc, and an empty space represents a square with no disc at all.

```
struct NaiveBoard {  
    board: [[char; 8]; 8] // in C, char[8][8]  
}
```

One may observe that each ordinary `char` is represented as a single byte (unsigned 8-bit integer) and thus this structure would take a total of 64 bytes in memory. While this spacial efficiency problem is egregious, I'd argue it's the least of its concerns.

The much larger problem with this datastructure is its impact on runtime performance of algorithms in terms of pure speed. While they *technically* have the same time complexity in Big  $O$ , Big  $O$  does not tell the full story here. This will be explained in further detail in the next section.

### 1.2 The Optimal Solution

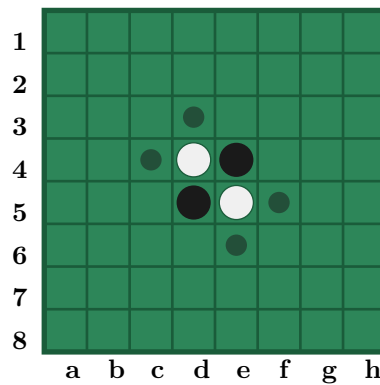
One may observe that each square in our board ultimately exists in one of three states at a time. These states are *has a black disc*, *has a white disc*, and *has no disc at all*. We could encode this information in just two bits! The first bit can encode whether or not there is a black disc while the second encodes whether or not there is a white disc. Naturally, if both bits are 0, then there is no disc at all.

So we'd have `00` for no disc, `10` for a black disc, and `01` for a white disc. `11` is not a valid state, so you would want to ensure your program cannot produce it. To create a full Othello board, you'll want to simply associate two unsigned 64-bit integers (one for black and one for white). I will hereafter refer to this structure as a *BitBoard*. An awesome thing about this representation is that we can derive certain properties using only bitwise operations.

For example, we can compute *all* of the empty spaces by computing the negation of the bitwise OR of the two boards. To do the same with the naive solution, would require that you loop through the datastructure. Since the dimensions of the structure are constant, that algorithm

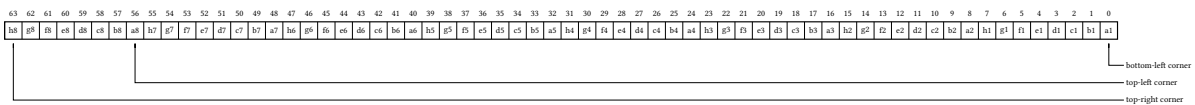
would have constant time complexity. They may be equivalent in a sense, but they still perform quite differently. The naive requires repeated randomly array-accessed string comparisons while our new algorithm is, depending on CPU architecture, executing only one or two instructions total.

Perhaps that operation is too simple to be fully representative. In the following chapters, I will demonstrate that same general bitwise approach can be had for all of the required algorithms.



**Figure 1:** Starting Position

At the bit level, the structure of a single sub-board is as follows:<sup>1</sup>



**Figure 2:** Layout for a single sub-board

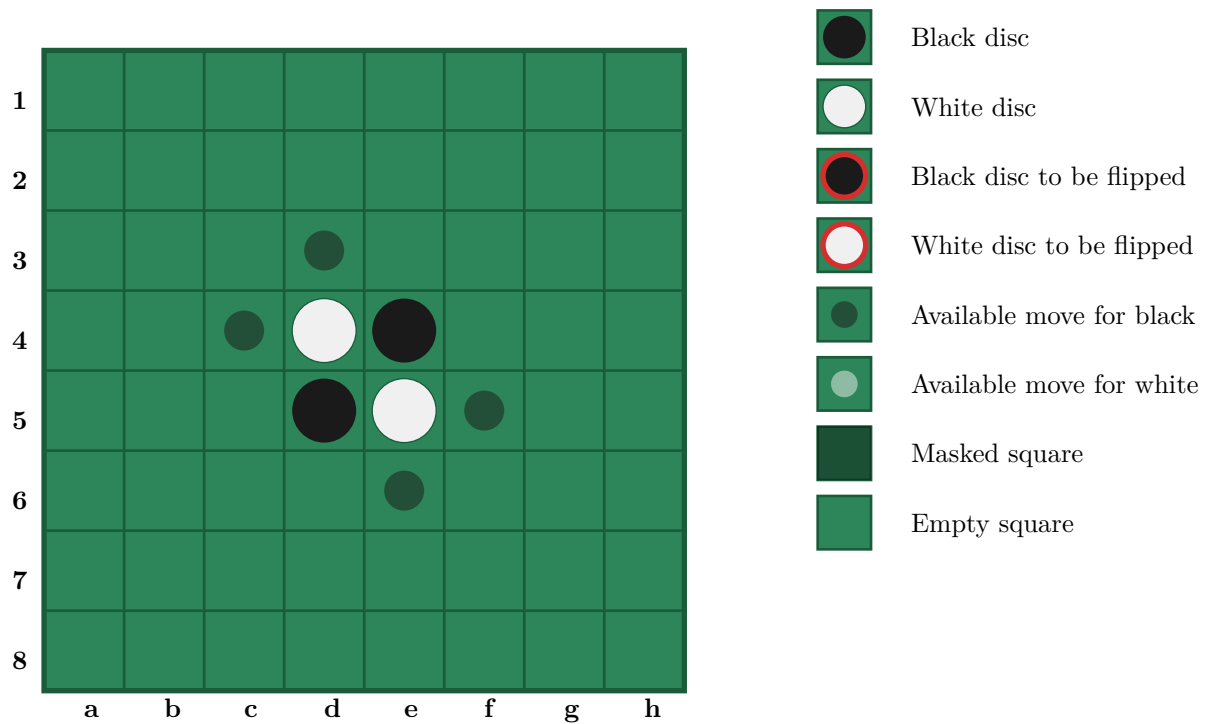
Now an entire game's board would simply be two of these. Ideally allocated contiguously in memory as part of one structure. My BitBoard is represented by the following rust code:

```
// `Board` is referred to as "sub-board" in this paper.
pub type Board = u64;
pub struct BitBoard {
    pub boards: [Board; 2],
}
```

### 1.3 A Note on Notation

I will use boards with descending rank and ascending file with the following legend. The starting board and black's first available moves will appear as follows:

<sup>1</sup>Text is a bit small. May have to zoom.



## 2 Board Operations

### 2.1 Generating Moves

In Othello, the only legal moves are ones that *flank* your opponent. In other words, you have to place a disc that pinches one or more of your opponent's pieces.

In simple terms, the algorithm is as follows:

```

1 keep a running sub-board of available moves
2 for each disc of the current player's discs
3   for each direction
4     keep a running sub-board of opponent squares in this direction
5     while the next square in that direction is has an opponent disc
6       | add to running sub-board of opponent squares
7     end
8     if next square in direction is empty
9       | add next empty square to running sub-board of moves
10    end
11  end
12 end

```

### 2.1.1 The Optimal Solution

To generate moves, we need to generate a sub-board (unsigned 64-bit integer) where bits are excited if and only if the current player can place a disc there.

Now, one might scratch their head wondering how they gain anything with this datastructure if you need to go disc by disc and shift them over repeatedly to determine whether or not there is an available move. Thankfully, because of the structure of the BitBoard, we can avoid this process entirely!

Now, since my structure happens to be Big Endian<sup>2</sup>, to compute a sub-board with every piece moved to the right, we can shift the entire board to the left by one. Observe that in our diagram, **b1** is to the *left* of **a1** despite it being to the *right* in the board visually<sup>3</sup>. With that in mind, we can just think of all the different values that correspond to particular adjacencies. To go up, trivially, we shift left by 8. To go up and to the right, we shift once more. To go up and to the left, we shift by one less. Going down, we mirror the shifts to go up but swap the diagonals. To go to the left, we shift to the right once. Thus, we have the following shift constants:


« 7	« 8	« 9
« 1		» 1
» 9	» 8	» 7

Figure 3: Shift constants

Consider a black disc on **h1**. If we shift it to the right, where does it end up? Observe what happens:

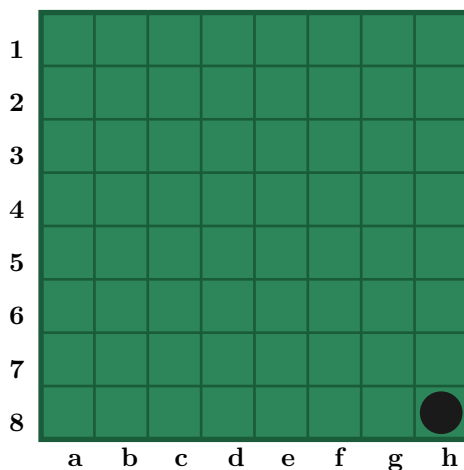


Figure 4: h1

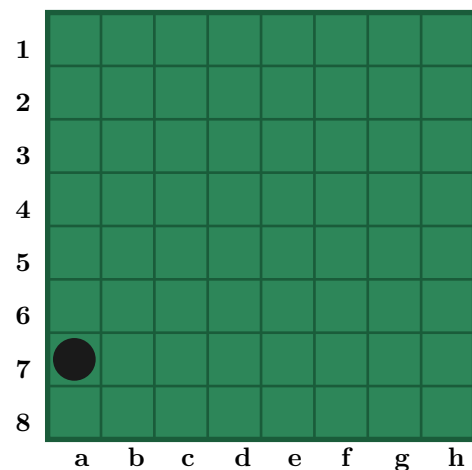
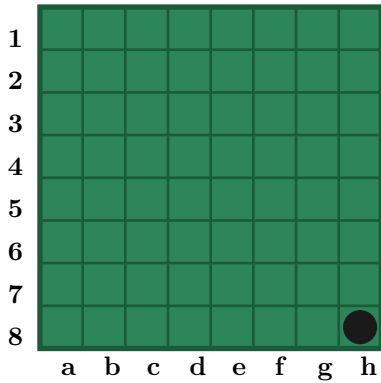
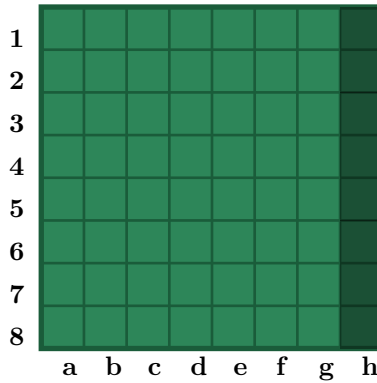
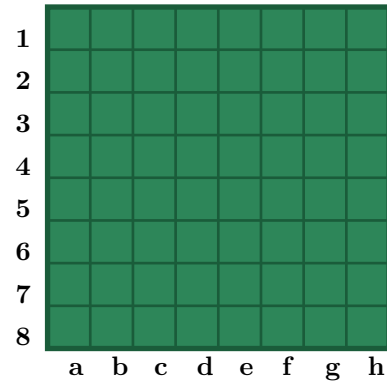


Figure 5: h1 << 1

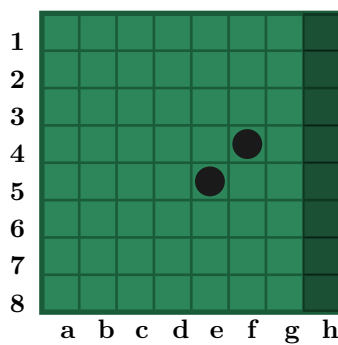
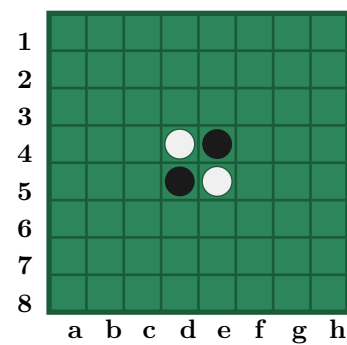
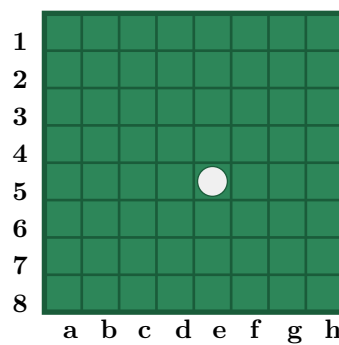
<sup>2</sup>It would be equally viable in Little Endian, but you would need to reverse directions of shifts.

<sup>3</sup>An advantage of a Little Endian implementation is that the shift logic would be slightly more intuitive. However, in a Big Endian structure, the sub-board containing only **a1** is defined as 1 which just seems natural to my human brain.

Therefore, it is prudent that we prevent wrap-around when moving horizontally or diagonally. We can do this by masking out the **a** file when moving left (visually) and masking out the **h** file when moving right.

Figure 6:  $!h$ Figure 7:  $(h1 \ \& \ !h) << 1$ 

Thus, we can associate directional shifts with masks. With those pieces in mind, let's go through the algorithm to find left-flanks in the starting position for Black<sup>4</sup>.

Figure 8:  $(B \ \& \ !h) << 1$  Figure 9:  $((P \ \& \ !h) << 1) \ \& \ O^5$ 

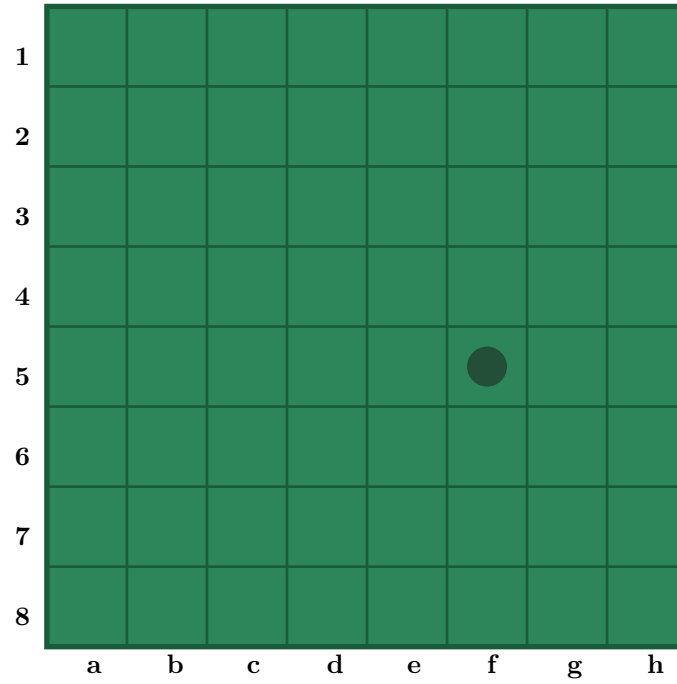
The result above is only the first pass when computing left flanks. You would want to run the same computation again six more times with the previous result as input instead of  $B$ . However, since there are no more discs to the right, we'd end up with the next seven passes will end up being effectively no-ops. Let's call this sub-board  $A_7$ .

Once we've finished looking seven squares to the right, we want to add<sup>6</sup> empty squares to the right of the squares we've computed to our total sub-board of available moves  $M$ .

<sup>4</sup>A left flank involves placing a disc to the right of an enemy disc

<sup>5</sup> $P$  is the current team playing, and  $O$  is the opponent.

<sup>6</sup>**XOR** is slightly faster and equivalent in this case.

Figure 10:  $((A_r \ll 1)E)$ 

,

Thus, we have determined that our sub-board  $M$  contains **f5**. Okay, with that out of the way, let's generalize the algorithm for each direction.

---

**Algorithm 1: Move Generation**


---

```

1:  ▷ Define the directional shift constants
2:   $D \leftarrow \{9, 8, 7, 1, -1, -7, -8, -9\}$ 
3:  ▷ Define the masks ( $F_a$  includes all but  $F_a$ )
4:   $F_a \leftarrow 0x7F7F7F7F7F7F7F7F$ 
5:   $F_h \leftarrow 0xFEFEFEFEFEFEFEFEFE$ 
6:   $all \leftarrow 0xFEFEFEFEFEFEFEFEFE$ 
7:   $M \leftarrow \{F_a, all, F_h, F_a, F_h, F_a, all, F_h, \}$ 
8:
9:  ▷ A helper function that shifts in the appropriate direction depending on the magnitude
10: procedure DIRECTIONAL_SHIFT(magnitude, board)
11:   if magnitude  $\geq 0$  then
12:     return board  $\ll$  magnitude
13:   end
14:   else
15:     return board  $\gg$  magnitude
16:   end
17: end
18:
19: procedure AVAILABLE(P, O)
20:   ▷ Define the board of empty squares
21:    $E \leftarrow !(P \mid O)$ 
22:

```

---



---

```

23: move ← 0
24: i ← 0
25: while i < 8 do
26:   d ← D[i]
27:   m ← M[i]
28:   A ← directional_shift((d, P&m))
29:   A ← directional_shift((d, A&m))
30:   A ← directional_shift((d, A&m))
31:   A ← directional_shift((d, A&m))
32:   A ← directional_shift((d, A&m))
33:   A ← directional_shift((d, A&m))
34:   A ← directional_shift((d, A&m))
35:   move ← move | A
36:   i ← i + 1
37: end
38: return move
39: end

```

---

If the algorithm is performed correctly, you will end up with the following board of legal moves:

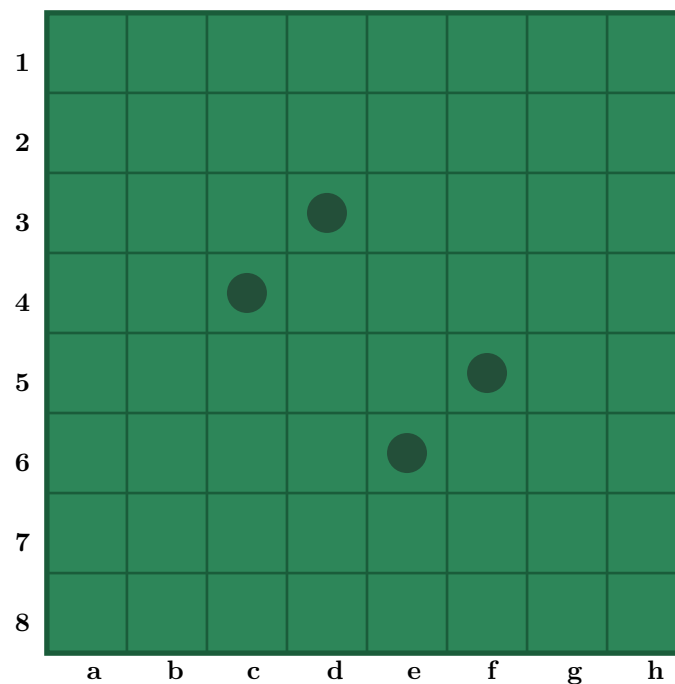


Figure 11: Starting moves for Black

### 2.1.2 Implementation in Rust

```

const NOT_A_FILE: Board = 0x7F7F7F7F7F7F7F7F;
const NOT_H_FILE: Board = 0xFEFEFEFEFEFEFEFE;
const SHIFT_MASK_COMBOS: [(i8, Board); 8] = [
    (9, NOT_A_FILE), // 9 (up right)
    (8, Board::MAX), // 8 (up)
    (7, NOT_H_FILE), // 7 (up left)

```

```

    (1, NOT_A_FILE), // 1 (right)
    (-1, NOT_H_FILE), // -1 (left)
    (-7, NOT_A_FILE), // -7 (down right)
    (-8, Board::MAX), // -8 (down)
    (-9, NOT_H_FILE), // -9 (down left)
];

impl BitBoard {
    pub fn available(&self, current_team: Team) -> Board {
        let empties = !(self.boards[0] | self.boards[1]);
        let mut moves = 0;
        let (team, opponent) = (
            self.boards[current_team as usize],
            self.boards[current_team.next() as usize],
        );

        for (shift, mask) in SHIFT_MASK_COMBOS {
            let mut sub_move = shift_in_direction(shift, team & mask) & opponent;
            sub_move |= shift_in_direction(shift, sub_move & mask) & opponent;
            sub_move |= shift_in_direction(shift, sub_move & mask) & opponent;
            sub_move |= shift_in_direction(shift, sub_move & mask) & opponent;
            sub_move |= shift_in_direction(shift, sub_move & mask) & opponent;
            sub_move = shift_in_direction(shift, sub_move & mask) & empties;
            moves |= sub_move;
        }

        moves
    }
}

```

## 2.2 Playing Moves

### 2.2.1 Benchmarks