

Othello Agent Technical Report

Kevin Corina

CSCI 312: Introduction to Artificial Intelligence

University of North Carolina at Asheville

December 3rd, 2019

Contents

1. Introduction	1
1.1. Context.....	1
1.2. Formatting.....	1
1.3. Prerequisites	1
2. Representation.....	1
2.1. Bitboards.....	1
2.1.1. Datatypes	2
2.1.2. Notation	2
2.1.3. Undefined Behavior	3
2.2. Othello board.....	3
2.3. Moves.....	4
2.4. Players.....	4
2.5. Games	4
2.6. Team	5
3. Major Functions	6
3.1. Move	6
3.2. Generating open edges.....	6
3.2.1. Smear method.....	6
3.2.2. Binary dilation	7
3.3. Acting on Individual on-bits of a Bitboard	7
3.3.1. Naïve, iterative.....	7
Informed search	8
Informed Jumps Counting Leading Zeroes.....	10
3.4. Search Algorithm.....	10
3.4.1. Alpha-beta.....	10
3.4.2. Evaluation Function	12
3.4.3. Timing.....	13
4. Testing.....	13
4.1. Test suite.....	13
4.2. Referee.....	14
4.3. Manual Debugging	14
5. Further reflection.....	14

5.1. Tournament results.....	14
5.2. Room for improvement	15
5.2.1. Decoupling	15
5.2.2. Single-responsibility classes	15
5.2.3. Profiling	15
6. References	15

1. Introduction

1.1. Context

The semester-long project for the Fall class of CSCI 312: Intro to Artificial Intelligence was to write an agent that played a game of Othello. We were given a list of commands it must accept, Othello's ruleset, and instructions to develop an agent that finds moves using an alpha-beta search (or a more sophisticated approach). Student's agents were judged by their ability to play a legal game, their performance in a class-wide tournament, and other criteria.

I chose to write my Othello agent in C++, first using MSVC v14, then GCC v8.3 when it finally registered that the agents would be running on the computer science department's Linux server.

This technical report describes the workings of my Othello agent and the process I followed to develop it. It also describes what I learned and what I will do moving forward. In addition to adding a separate section for this, I have chosen to point out aspects I would approach in place as they're analyzed.

1.2. Formatting

Code, pseudocode, and filenames are printed in monospace with one indent:

```
int x = 0;
```

Production code pulled directly from the project has its source file at the top of the block. Preprocessing directives like headers are not included unless relevant to the example. Comments and whitespace may also be modified or removed entirely based on their relevance:

```
//from CommandLineInterface.h

const std::string cli::nextNonComment() {
    std::string input;
    do {
        std::getline(std::cin, input);
    } while (input.at(0) == 'C');
    return input;
}
```

In most cases I have chosen to forgo the ellipses that would suggest that there is code surrounding a snippet. Any given example should be presumed to be a snippet that cannot compile alone.

1.3. Prerequisites

It is assumed that the reader of this document is familiar with the game Othello and has a working knowledge of C++.

2. Representation

2.1. Bitboards

I chose to represent pieces using a bitboard. The most significant advantage is that certain operations can be performed very quickly. Adding or removing any number of pieces, comparing boards, and copying boards can each be done with a single bitwise operation.

2.1.1. Datatypes

The advantages of a bitboard rely on it being stored in memory as a 64-bit integer. In C++, extra care needs to be taken to ensure the correct data type is used. The intuitive choice in C++ is

```
long long unsigned int
```

But it turns out this is not *guaranteed* to be 64 bits. Implementations vary from platform to platform. I ended up using `uint64_t` from the `cstdint.h` header instead. It's an unsigned integer datatype guaranteed to be exactly 64 bits. I also opted to explicitly reference the `std` namespace in my code:

```
#include <cstdint.h>
```

```
std::uint64_t = 0;
```

Though it's not the most critical consideration for this project, when directly accessing `uint64_t` through `std` it is both guaranteed to exist and less likely to have its functionality overridden by other libraries.

My code is littered with `std::uint64_t`. I could have avoided typing it over and over again using using a `typedef`. This would also make the code more intuitive to read and interpret.

```
#include <cstdint.h>
```

```
typedef std::uint64_t bitboard;  
bitboard pieces = 0;
```

2.1.2. Notation

I refer to individual cells on a bitboard four different ways. As raw integers with a single on-bit, as a bit-index, as an x-y notation, and a-1 notation. (Note that the x-y coordinate has an origin of 0-0 and a-1 notation has an origin of a-1 and spans from a to h on the horizontal axis.)

This variation introduces an unfortunate complexity and hinders development. However, each notation also has its own advantages in different situations. I ultimately decided that the added complexity was worth it and made an attempt to keep any names and conversions clear in my code.

I also generated two lookup tables to switch between x-y and bit-index

```
//From BitboardHelper.h
```

```
constexpr int XY_TO_BI[8][8] = { { 0 ,8 ,16 ,24 ,32 ,40 ,48 ,56 }, {1, 9, 17,  
25, 33, 41, 49, 57 }, {2, 10, 18, 26, 34, 42, 50, 58 }, {3, 11, 19, 27, 35, 43,  
51, 59 }, {4, 12, 20, 28, 36, 44, 52, 60 }, {5, 13, 21, 29, 37, 45, 53, 61 },  
{6, 14, 22, 30, 38, 46, 54, 62 }, {7, 15, 23, 31, 39, 47, 55, 63 } };
```

```
constexpr int BI_TO_XY[64][2] = { {0, 0},{1, 0},{2, 0},{3, 0},{4, 0},{5,  
0},{6, 0},{7, 0},{0, 1},{1, 1},{2, 1},{3, 1},{4, 1},{5, 1},{6, 1},{7, 1},{0,  
2},{1, 2},{2, 2},{3, 2},{4, 2},{5, 2},{6, 2},{7, 2},{0, 3},{1, 3},{2, 3},{3,  
3},{4, 3},{5, 3},{6, 3},{7, 3},{0, 4},{1, 4},{2, 4},{3, 4},{4, 4},{5, 4},{6,  
4},{7, 4},{0, 5},{1, 5},{2, 5},{3, 5},{4, 5},{5, 5},{6, 5},{7, 5},{0, 6},{1,  
6},{2, 6},{3, 6},{4, 6},{5, 6},{6, 6},{7, 6},{0, 7},{1, 7},{2, 7},{3, 7},{4,  
7},{5, 7},{6, 7},{7, 7} };
```

and two functions to change an x value to a-1 notation

```
//From BitboardHelper.cpp

const char bbh::xToColumn(int x) {
    return (char)('a' + x);
}

const int bbh::columnToX(char col) {
    return (int)(col - 'a');
}
```

2.1.3. Undefined Behavior

When trying to do math with bitboards and small literals I found that I was getting strange errors and seemingly random data in my bitboards.

I eventually realized that literal integers in my C++ source code were not compiling as 64-bit unsigned integers. This was error occurring because shifting the default int by the much larger std::uint64_t was undefined behavior in my compiler. The workaround was to explicitly cast the literal to std::uint64_t when I defined it. For example, if I wanted to add a piece to a bitboard bb at the 27th bit, the behavior of the following code would be undefined and unpredictable

```
bb |= 1 << 27;
```

where the behavior of this code would be defined and predictable

```
bb |= (std::uint64_t 1) << 27;
```

I wanted to avoid typing (std::uint64_t 1) over and over again and creating a constant ONE

```
constexpr std::uint64_t ONE = 1;
bb |= ONE << 27;
```

However, I didn't actually end up using the ONE constant that much in my code. If I were to do it again, I'd use the first workaround just to have one less global variable.

2.2. Othello board

The OthelloBoard stores the position of all pieces. Because the state of a bitboard cell is binary, the placement of both player's pieces must be stored in two separate bitboards, each stored in an array with a Color as its key. This actually worked alright

```
//from OthelloBoard.h

std::uint64_t pieces[2] = { 0,0 };
std::uint64_t empty = 18446744073709551615;
```

The OthelloBoard is responsible for applying moves and keeping track of the number of pieces on the board. While it is somewhat redundant, I chose to also maintain a bitboard of all empty cells so I wouldn't have to calculate it so frequently.

It also generates the legal moves for any given player.

All of these responsibilities make for a far too cluttered class. Going forward I will be making a greater effort to observe the “single responsibility” principle.

2.3. Moves

Moves are represented by the struct Move

```
//from Move.h

struct Move {
    //the bit-index of the piece to be placed
    int bi = 0;

    //the number of pieces captured by this placement
    int numOfPieces = 0;

    //bitboard of enemy pieces that will be flipped when this move is made
    std::uint64_t toFlip = 0;

    //score given by an evaluation function
    double score = 0;

    Move();
    Move(int bi, int numOfPieces, std::uint64_t toFlip, double score);
    static Move noScoreMoveFromXY(int x, int y, int numOfPieces, uint64_t
        toFlip = 0);
};
```

The Move class, too, is bloated. It tries to do too much. Those extra constructors were thrown in hastily and without thought while I was writing tests that needed dummy-moves.

2.4. Players

The Player class is an attempt to apply the “Open for Extension, Closed for Modification” principle. It’s a virtual class, essentially an interface but with a few protected variables and a constructor. Because both AlphaBetaAgent and CommandLinePlayer extend Player, an OthelloGame object doesn’t ever need to know anything about the player’s implementation and gives me the flexibility to develop it relatively independently.

2.5. Games

A match is overseen by an OthelloGame object. The match is started by calling startAlphaBetaAgentVersusCommandLineOpponentGame() in Gametypes.h.

The clunky name and extraneous class are left over from abandoned attempts to make the different game types more robust and modular.

In the typical game between my bot and another student’s, the function gets the color of the local bot, creates an OthelloBoard, CommandLineOpponent, and AlphaBetaAgent object. Then, it passes pointers to an OthelloGame constructor. After the local bot has signaled ready the game is started by calling OthelloGame::start()

The game loop looks something like this in pseudocode:

```

Start():
    running = true
    currentPlayer = black player
    While running:
        prompt for the currentPlayer's next move
        If the move isn't a pass:
            apply the move the OthelloBoard
            update the number of pieces
            notify the current player's opponent what the move was
        currentPlayer = the opponent of the currentPlayer

```

I found that printing the black score `n` when another player made an illegal move or passed twice triggered a loss for me, so I stopped trying to keep track of both and trusted the Referee to address it. This worked fine.

I'm actually pretty fond of how `OthelloGame` turned out, but I foiled my own attempts to keep systems decoupled when I declared the `PASS` constant in `OthelloGame`. Now, all players have a dependency on `OthelloGame` in order to generate moves.

2.6. Team

It was recommended that I use integers 0 and 1 for teams. I wanted something that would be safer than passing an integer around, so I tried to wrap teams in an enum `Color`.

```

//from Color.h

enum Color { BLACK, WHITE };

```

I then wanted to abstract out some repetitive color code, like determining a `Color`'s name or its opposite color. It seemed unnecessary to give it its own class. I didn't want the extra overhead of copying/passing an object around. I settled for making the common functions static and wrapping them in a struct for organizational purposes.

```

//from Color.h

#include <string>
#ifndef COLOR_H
#define COLOR_H

struct ColorInfo {
    static const char colorChar[2];
    static const Color opponent[2];
    static const std::string colorName[2];
};

typedef ColorInfo ci;

#endif

```

I made a shorter typedef for `ColorInfo` and instructed the compiler to only define the contents of the file once. This effectively made it another global variable. While I still appreciate the readability it offers in some places, the color implementation mostly served to complicate my code and slow my program.

3. Major Functions

3.1. Move

`OthelloBoard::makeMove()` takes a `Move` object and a `Color` then applies it to the black and white bitboards. It calls `flip()` and `place()` then returns the number of pieces changed.

```
//from OthelloBoard.cpp

int OthelloBoard::makeMove(Move move, Color player) {
    if (move.numOfPieces != 0) {
        flip(move.toFlip, move.numOfPieces, player);
        place(move.bi, player);
    }
    return move.numOfPieces;
}
```

`flip()` uses a `Move` object's `ToFlip` bitboard to effectively swap the corresponding bits in the local bitboards of the player and their opponent, then and update the number pieces of each.

```
//from OthelloBoard.cpp

void OthelloBoard::flip(std::uint64_t toFlip, int flipCount, Color player) {

    pieces[player] |= toFlip;
    pieces[ci::opponent[player]] &= ~toFlip;
    numOfPieces[player] += flipCount;
    numOfPieces[ci::opponent[player]] -= flipCount;

}
```

`place()` just turns on the player's board's bit at the bit-index of the move and increments the number of player pieces.

```
//from OthelloBoard.cpp

void OthelloBoard::place(int pos, Color c) {
    pieces[c] |= bbh::ONE << pos;
    empty &= ~(bbh::ONE << pos);
    numOfPieces[c] += 1;

}
```


3.2. Generating open edges

3.2.1. Smear method

To find a board's open edges, I "smear" the enemy board by combining the results of shifting it 1 cell in each of the 8 directions with bitwise-or operators.

Up and down are intuitive. I just shift the board 8 bits left or 8 bits right. The extra over/underflow on the top and bottom edges is lost. To get the left and right smear I shift the board 1 bit left and 1 bit right, but I need to cull the overflow on the rightmost or leftmost edges. After that, it's a matter of clearing any occupied spaces using a bitwise-and on the bitboard of empty spaces. Diagonals follow a similar process and each diagonal needs only a single shift. I can determine the shift value and direction by

combining the shift values of the corresponding horizontal and vertical translations (treating left shifts as negative and right shifts as positive).

<<9	<<8	<<7
<<1		>>1
>>7	>>8	>>9

I chose to put this into C++ macro to isolate it without introducing the extra overhead of a function call. The code here is reformatted to make it more readable (this macro would will not compile in context):

```
//from OthelloBoard.cpp

#define OPEN_EDGES(p)
    empty & (
        p << 8 |
        p >> 8 |
        (bbh::NOT_RIGHT_EDGE & (p << 1 | p >> 7 | p << 9)) |
        (bbh::NOT_LEFT_EDGE & (p >> 1 | p << 7 | p >> 9))
    )
;
```

3.2.2. Binary dilation

This method was brought to my attention by Ben Nicholas, and I was intrigued enough to explore further.

Binary dilation has origins in low-level graphics programming. Unfortunately, it didn't seem particularly practical without direct access to the GPU. My attempts to implement it took far more operations than the smear method by requiring me to superimpose a 3x3 matrix on each of the 64 bits.

One way this might be made faster is by also storing the locations of each piece in a list parallel to the a bitboard. I didn't explore this further because I thought that building, passing, updating, and copying that list could get too slow and defeat the purpose of using a bitboard.

I considered trying to get around this by superimposing a 3x3 grid on a separate edge-board every time a piece is placed. This would make placing and flipping slower and more complicated, but it could nearly eliminate edge-generation if done right. I ultimately decided to go for the smear method and decided that re-writing my code to use binary dilation without access to the GPU would not be worth the time or complexity. In retrospect, this would be a good idea to revisit.

3.3. Acting on Individual on-bits of a Bitboard

3.3.1. Naïve, iterative

In order to determine whether the open edges generated were valid I chose to analyze potential move individually. Efficiently getting the bit-indices of the on-bits in a bitboard turned out to be particularly difficult.

The naïve approach was to use a for-loop from 0 to 63, then for each iteration create a raw bitboard of at the index of the counter to check if that cell was checked, and finally do something at that cell.

```
std::uint64_t bb;
```

```

for (int i = 0; i < 64; i++) {
    if ((bbh::ONE << i) & bb) {
        //do something
    }
}

```

In an attempt to optimize this code, I changed the for-loop to a while-loop. I combined the counter and raw bitboard to a single bitboard that's shifted left by 1 bit each iteration.

```

std::uint64_t TOP_LEFT = 9223372036854775808;

/*10000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000*/

std::uint64_t bb;

for (uint64_t pos = 1; pos <= TOP_LEFT; pos << 1) {
    if (pos & bb) {
        //do something
    }
}

```

While its runtime is $\Theta(1)$, this snippet still leaves a lot to be desired. No matter the number of pieces on the board it requires 64 shifts, 64 bitwise-and's, and 64 inequality checks.

In fact, it may be even worse than the for-loop because the exact number of iterations isn't immediately obvious to the compiler. Where the compiler can unroll the for-loop 64 times, this won't be unrolled and therefore requires extra comparison and switch operations.

Informed search

Instead of checking every single bit individually I tried to quickly find the on-bits and act only on those.

3.3.2. Lookup tables for bit-indices

The first thought that popped into my head was generating a lookup table to find the indices of the on-bits from a bitboard. I would pass the table a bitboard, it would return an array of bit-indices corresponding the on-bits. Obviously, the comprehensive table of all possible configuration would be too large to use.

I ended up doing the math anyway. Such a table would need 64^2 arrays for every possible configuration varying from 1 to 64 chars long (char is the smallest possible C++ datatype that goes up 64, occupying only 1 byte versus than the 2 bytes of a short int), one. The total bytes s of the table in memory would end up being

$$2 \sum_{n=0}^{64} \frac{64!}{(64-n)!} = s \text{ bytes}$$

...or ~689828884059608166089493852753613138094818430030937180678910890468 *yottabytes*.

I did not take this approach.

I instead explored the more rational tactic using a lookup table for every possible configuration of an 8-bit row. Slightly modifying the earlier equation, I determined the size of this new array before attempting to generate the data

$$2 \sum_{n=0}^8 \frac{8!}{(8-n)!} = 219202 \text{ bytes}$$

219 kilobytes isn't too much in the grand scheme of things, so I ended up trying it. The first implementation looked something like this in pseudocode:

```
char[][] rowLookup = [[], [0], [1], [0,1], [2], ..., [0,1,2,3,4,5,6,7]]

//e.g. rowLookup[97] == [0,1,7] because
//97 is represented as 1 1 0 0 0 0 1 in binary
//                ^ ^      ^
//                0 1      7

bitboard bb
onbits = []

//((bb >> 8 * y) % 256 isolates row y
//rowLookup[(bb >> 8 * y) % 256] returns the bit-index of each on-bit in
//a row shifted left by 8y

for i = 56, i >= 0, i >> 1:
    onbits[(bb >> i) % 256]
    for j = 0, j < onbits.size, j++:
        bi = onbits[j] << i
        //do something here
```

I then realized that I might be able to only store only the index of the next on bit of the bitboard, shift the bitboard right that many bits, and repeat until the bitboard was empty. A table for each row would only be 2^9 bytes. In pseudocode:

```
//empty row is 8
char[] rowLookup = [[8], [7], [6], [6], ..., [0],[0],[0],[0]]
//e.g. rowLookup[47] == 2 because
//47 is represented as 0 0 1 0 1 1 1 in binary
//                ^
//                --->2 leading zeros

int bi = 63
bitboard bb
int nextOn = 0
```

```

//keep shifting right until bb is gone
while bb ^ 0:
    nextOn = rowLookup[bb % 256]
    pos -= nextOn
    bb >> nextOn
    if nextOn ^ 8:
        //do something

```

3.3.3. Informed Jumps Counting Leading Zeroes

It turns out that this “next on-bit” operation has a real name in a lot of compilers: “count leading zeroes.” I finally stumbled upon GCC’s built-in function for just it - `__builtin_clzll(bb)`. Using this I was able to forgo the if statement of the previous and approach and find the next on-bit using an optimized, low-level instruction.

```

int i = 0;
int nextOn = 0;
int pos = 64;
std::uint64_t bb;

while ((nextOn = __builtin_clzll(bb) + 1) ^ 64) {
    pos -= nextOn;
    //do something
    bb <<= nextOn;
    i++;
}

```

While this can be faster on a sparse bitboard, on a dense board the naïve approach may actually be much faster. For a bitboard with n pieces on it the code performs n bitwise-xor operations, n built-in calls, n shifts, 3 assignments, $2n$ increments, and n decrement. Compare this to the flat **asdasda** operations for the naïve approach. Making the relatively bold assumption that each of these operations takes roughly the same amount of time, this would mean that the “tipping point” number m of on-bits can be found using simple algebra.

$$m = 64 * 3 = 6n + 3$$

$$m \approx 32$$

Suggesting that it might be better to only use this function when the number of on-bits of a board is 32 or fewer.

I spent way too much time on this problem, trying to be clever. It wasn’t a significant bottleneck, the advantage of the “optimized solution” is dubious, and I ultimately had to revert to the iterative approach from my code last minute because I noticed some edge cases where the others always failed.

3.4. Search Algorithm

3.4.1. Alpha-beta

I used the standard Alpha-beta algorithm. I adapted it from that given on the Moodle page.

For every iteration it clones the OthelloBoard, applies the move, then performs the ‘opposite’ search (maximizing v. minimizing) on the new board.

```

Move AlphaBetaAgent::alphaBetaSearch(OthelloBoard currentBoard, Move lastMove,
Color player, double alpha, double beta, int depth, bool maximizing) {

    //When it hits the maximum depth, a full board, or the exceeds the time
    //limit, the evaluation function is performed and the score returned.
    if (depth == 0 || currentBoard.empty == 0 || time(NULL) >= searchLimit) {
        lastMove.score = evaluateBoard(&currentBoard);
        return lastMove;
    }
    else {
        //It's a real pain in C++ to call a function that returns an array,
        //so instead the agent passes an array by reference and has it filled.
        Move moves[64];
        int size;
        currentBoard.generateLegalMoves(player, moves, size);

        //if there are no legal moves, then it adds pass to the list.
        if (size == 0) {
            moves[0] = Move(OthelloGame::PASS_BI_FLAG, 0, 0, DBL_MIN);
            size = 1;
        }

        //everything that follows is standard AB search
        int bestMoveIndex = 0;
        if (maximizing) {
            for (int i = 0; i < size; i++) {
                OthelloBoard nextBoard = currentBoard;
                nextBoard.makeMove(moves[i], player);
                moves[i].score = alphaBetaSearch(nextBoard, moves[i],
                    ci::opponent[player], alpha, beta, depth - 1,
                    false).score;

                if (moves[bestMoveIndex].score < moves[i].score) {
                    bestMoveIndex = i;
                }
                if (moves[bestMoveIndex].score >= beta)
                    break;
                alpha = std::max(alpha, moves[bestMoveIndex].score);
            }
        }
        else {
            for (int i = 0; i < size; i++) {
                OthelloBoard nextBoard = currentBoard;
                nextBoard.makeMove(moves[i], player);
                moves[i].score = alphaBetaSearch(nextBoard, moves[i],
                    ci::opponent[player], alpha, beta, depth - 1,
                    true).score;
                if (moves[bestMoveIndex].score > moves[i].score) {
                    bestMoveIndex = i;
                }
                if (moves[bestMoveIndex].score <= alpha)
                    break;
                beta = std::min(beta, moves[bestMoveIndex].score);
            }
        }
    }
}

```

```

        }
    }
    return moves[bestMoveIndex];
}
}

```

The next step to make this better would be to wrap some of the redundant code in a macro, increasing the readability without introducing additional overhead.

On top of that, cloning a the bitboard every single time is a waste. Creating new objects is expensive, even small ones. It would be so easy to implement the ability to undo a move with just a few operations. For example:

```

int OthelloBoard::undoMove(Move move, Color player) {

    //un-flip the opponents pieces
    pieces[ci::opponent[player]] |= toFlip;

    //re-flip the players pieces
    pieces[player] &= ~toFlip;

    //remove the piece placed
    pieces[player] ^= bbh::ONE << pos;

    //reduce the player's counter and give the points back to the opponent
    numOfPieces[player] -= flipCount - 1;
    numOfPieces[ci::opponent[player]] += flipCount;
    empty |= bbh::ONE << move.bi;

    return move.numOfPieces;
}

```

If I had done this, I would be able to basically backtrack with each recursion without ever creating a new, redundant board.

It would also improve my searches so much If I actually implement iterative deepening. As of now, it's entirely possible that the program could timeout after searching to deep in a single move's tree rather than doing a shallower search exploring multiple moves.

3.4.2. Evaluation Function

The primitive evaluation function analyzes only one piece of data: The ratio of the difference of player pieces and opponent pieces to the total number of pieces.

$$\frac{\text{player pieces} - \text{opponent pieces}}{\text{player pieces} + \text{opponent piece}}$$

```

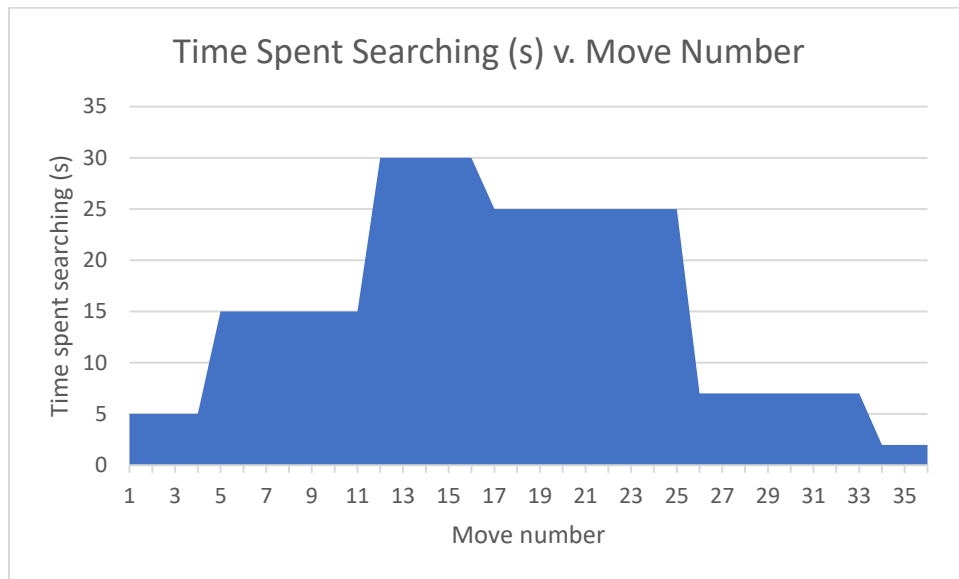
double AlphaBetaAgent::evaluateBoard(OthelloBoard* toEvaluate) {
    return
        (double(toEvaluate->getNumOfPieces(color)) -
         (double(toEvaluate->getNumOfPieces(ci::opponent[color])))) /
        (double(toEvaluate->getNumOfPieces(color)) +
         double(toEvaluate->getNumOfPieces(ci::opponent[color])));
}

```

I am not content with this evaluation function. If I could do it again, I would write something more advanced that weighted certain cells (like the corners) and considered the number of moves my opponent had in that state.

3.4.3. Timing

Every turn, the AlphaBetaAgent is allocated a pre-determined number of seconds to search. After that time elapses it stops the search and recurses upward. I chose to allocate less time to the earliest and latest searches, and more time to the middle searches.



The timing was guesswork. I did not apply any sophisticated analysis; I only acknowledged the general arc of the game explained by my professor and peers then picked some numbers by hand.

I could also *double* the search time if I managed to thread the agent. This would allow me to search while the opponent is searching, a huge advantage to have over many of the un-threaded bots.

4. Testing

4.1. Test suite

I did follow some tenants of TDD, but to call my method of testing a “suite” may be overstating it. If I wanted to test my Othello bot I typically ran the `bool testOthelloBoard()` method of `OthelloBoardTests.cpp`. The `OthelloBoard` has just about 90% of all the moving parts in the system.

```
//from OthelloBoardTests.cpp

bool testOthelloBoard() {
    cli::comment("Othello version 2.3");
    testOthelloBoardConstructors();
    testPlace();
    testFlip();
    testEdgeGen();
    testFlipGen();
    testMoveGen();
    testMove();
}
```



```
    return true;  
}
```

These tests are lightweight and far from comprehensive, but they touch on nearly all the fundamental components. This made development much easier. Almost every time I broke something, I would know it immediately. While plenty of issues wouldn't become clear until after I had played a full game using the Referee, this caught most of them.

I absolutely could have chosen to be more thorough in my testing, but looking back only tests I regret not making are ones for my agent's alpha-beta search. I deemed it too tedious to bother writing in comparison to the potential benefits. This was very wrong. Debugging the alpha-beta search was a huge pain and many problems went unnoticed for too long.

4.2. Referee

I had some issues getting the Referee to work. Even after following the instructions, I found myself getting an error about limited memory space. I ultimately set up my own local virtual machine and got it working there. Waiting for 10 to 20 minute games to finish was a slow but necessary process.

4.3. Manual Debugging

When debugging I used a number of tools. The most useful were the function to convert a bitboard to a string I could comment to the command line, and the function to create a bitboard from a human readable string.

After running this with a visualizer I realized just how slow the conversion from bitboard to string was, computationally. I removed the function for production but found it a godsend when developing.

5. Further reflection

5.1. Tournament results

I downloaded all of logs generated by the Referee into a single .txt file and wrote a small Python3 script to gather win/loss data for every agent. The script was not thoroughly debugged and just hacked together to get a general idea of how I placed. Unfortunately, I have since lost the file.

If I did my math right, I had a win to loss ratio of about .66. This put me just above the middle of the pack. I am pleasantly surprised. Because I didn't work out how to use the Referee effectively until late in the semester, I had been uncertain of my bot's status for much of the project.

While there are serious bottle necks and many glaring opportunities to optimize my code, my greatest advantage here was still speed. My C++ code and low-level bit-wise implementations will typically search much faster (and therefore deeper) than my opponents' Java code can.

My greatest weakness is my evaluation function. As evidenced by my placement, there are still cases a better search algorithm simply beats a deeper search, where my opponent's code is also written in a low-level language. The evaluation function and other aspects of my code to improve are explored below.

5.2. Room for improvement

5.2.1. Decoupling

Despite going into this project with the best intentions, many of my classes have unnecessary references to other classes implementation. This makes my code tangled and at times difficult to parse. Instead of programming to an interface, I exposed data and introduced too many dependencies.

5.2.2. Single-responsibility classes

My classes, such as Color and OthelloBoard, frequently try to do too many things. Abstracting out a few classes to isolate certain functions would be a fairly simple process, we my code less coupled.

5.2.3. Profiling

In future projects I will be making greater use of the profiler. By using it more frequently I would be able to make better, more informed choices on what to optimize and how. It is terrible practice to try to just intuit what is bottleneck and better prioritize. Because I didn't use the profiler effectively I spent far too much time fiddling with less important code.

6. References

1. S. Russel and P. Norvig, *Artificial Intelligence: A modern Approach*, 3rd ed. Upper Saddle River, NJ: Pearson education, 2010.