Designing An Intelligent Othello System Using Turn-based Partitioning

Rodney Shaghoulian

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801-2302 USA

August 2016

Abstract

In this paper, the theory behind designing an advanced AI to play the 2-player board game Othello is described and augmented. We begin by providing various bitboard techniques. Minimax Search is used as the foundation of our game tree search, and Alpha-beta pruning and move order selection are further added to increase efficiency. Search is cut off at a depth limit and evaluation functions are used to determine board utilities. The idea of turn-based partitioning is introduced as an improvement that can be applied to evaluation functions. Experimental results of incorporating turn-based partitioning show increases in win percentages against a benchmark opponent.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Background

Othello is a 2-player board game. It was invented in 1883 and was originally called “Reversi”. For comparison, Chess originated in approximately the 6th century, and Checkers originated in the 12th century. This context makes Othello a relatively new and less studied game. However, its popularity has been steadily rising, and over 40 million physical copies of the game have been sold to date. The World Othello Championship originated in 1977 and remains popular today, as Othello is played in over 100 nations.

Figure 1: Othello Board

The beauty of Othello is in the simplicity of the rules. However, these simple rules lead to interesting complexities which have entertained and challenged players for over 100 years. This complexity that has surprisingly risen from simplicity has led Othello to coin the phrase “A minute to learn... a lifetime to master.”

1.2 Motive

Designing intelligent AI for board games has been a goal of computer scientists for decades. The original intention for such AI was to enable a human player to play against another (computer) player when no other human players were available. The first notable iterations of computer AI were for Chess and provided an endless amount of entertainment in 1-player mode for Chess enthusiasts. As technology progressed, so did the strength of the computer AIs. Faster processors allowed for more calculations and thus *smarter* AIs. The study of artificial intelligence sparked great research interest and began to emerge as a field. Different algorithms were developed to guide a computer’s intelligence and such algorithms are still being created and improved today. In May 1997, the Deep Blue Chess-playing system defeated the reigning world-champion Gary Kasparov in a six-game match [1]. In March 2016, Google’s AlphaGo system defeated the world-renowned Go player Lee Sedol in 4 out of 5 games [2]. Although these systems were state of the art in their time, there is always room for improvement (assuming the board game has not yet been solved).

Some games such as Checkers are too simple and do not provide much challenge for a computer player. Checkers was solved in 2007 in the sense that a computer can play as optimally as possible against a human opponent. Othello's search space greatly exceeds that of Checkers' search space and remains an unsolved game. This provides an excellent challenge for computer scientists. How can we design a computer AI that plays as close to optimal as possible using whatever tricks, game knowledge, and computation power that we do have? Designing such a system is our goal, and the *tbPony* Othello system was created in conjunction with this study.

2 Rules of the game

Othello is a 2-player game played on an 8x8 board similar to Chess and Checkers, with the exception that the board does not have a checkered pattern. There are 64 circular disks that are black on one side, and white on the other. The board starts in the configuration shown in Figure 2.

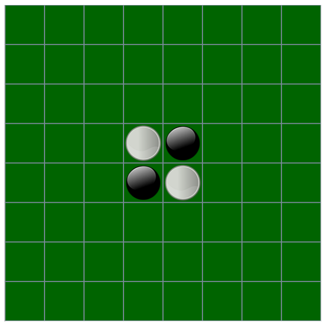


Figure 2: Initial Configuration

2.1 Player Turn: Placing A Disk

Each player gets 30 disks. Players alternate turns, and black is the first to play. On each turn, a player may place one disk on the board. The newly placed disk must "flank" one or more of the opponents disks. For black player, the newly placed piece must be placed next to a white disk in such a way as to create a line segment (horizontally, vertically, or diagonally) where the two endpoints are black disks and there are one or more white disks in between these endpoints. In this case, black flanks white's disk(s) and therefore captures them. All of the captured white disks must be flipped over to black disks. Flanking happens in all 8 directions simultaneously. The same rules apply to white player. Sample valid moves for black are highlighted light-green in 2 different board scenarios in Figure 3.

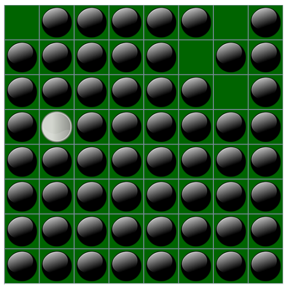
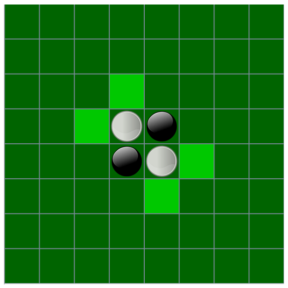
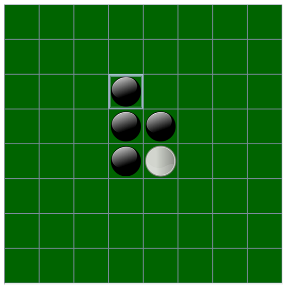
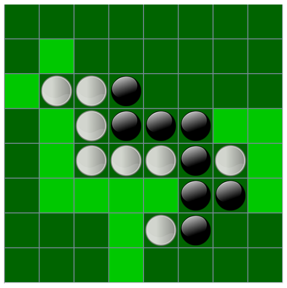


Figure 4: Neither player has a valid move. Black wins 59-1.

If a player has no valid moves, their turn is skipped. If neither player has any valid moves, the game ends. This can possibly occur before the entire board is occupied with disks, as shown in Figure 4. The winner of the game is whichever player has the most of their colored disks on the board when the game ends.


(a) Black's potential moves at start of game


(b) Board after capturing disks


(c) Another example of black's potential moves

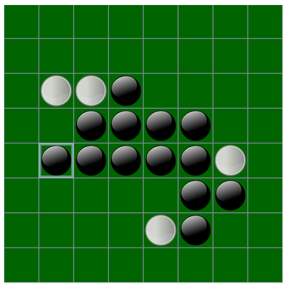
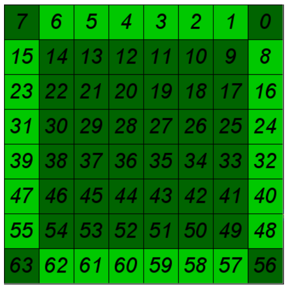

(d) Board after capturing disks

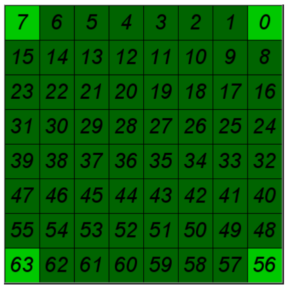
Figure 3: Potential moves for black player

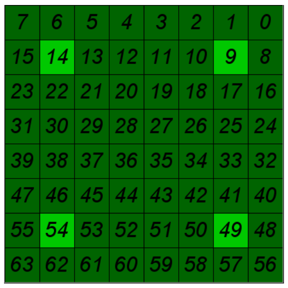
3 Basic Strategies

3.1 Stable Disks

In general, a disk can be flipped by surrounding it with disks of a different color in 4 possible directions: horizontally, vertically, on the NorthEast-SouthWest diagonal, or on the NorthWest-SouthEast diagonal. Any disk that cannot be flipped is referred to as a stable disk. A player should strive to obtain as many stable disks as possible, as these disks will necessarily contribute to the player's score at the end of a game.


(a) Edges


(b) Corners


(c) X squares

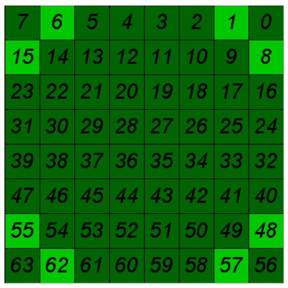

(d) C squares

Figure 5: Types of Squares

3.2 Edges

The squares referred to as edges are shown in Figure 5a. The disks in these squares are less likely to be flipped, as they can only be flipped in one direction as opposed to four directions for disks in the center of the board. For example, a disk on the left edge can only be flipped vertically, as that is the only way to surround the disk on both sides. Similarly, a disk on the top edge can only be flipped horizontally. Edges may seem like very powerful squares at first glance, but a disk on an edge is only stable if it is connected to a corner square with a disk of the same color.

3.3 Corners

Corner squares are very special squares. Only four corner squares exist (Figure 5b). When a disk is placed in a corner square, it immediately becomes stable since it cannot possibly be flipped in any of the four directions. Another advantage of owning a disk on a corner square is that it converts all of a player's disks that are adjacent to a corner square (horizontally or vertically) into stable disks. For example, if a black player has disks on squares 6, 7, 15, and 23, all four disks are stable as there is no way for white player to flip these disks. For these reasons, one of the primary goals for a competitive Othello player is to strive to place disks in as many corner squares as possible.

3.4 X squares and C squares

X squares (Figure 5c) and C squares (Figure 5d) are squares that should be avoided in most situations. This is because they give the opponent easy access to the much coveted corner squares. For example, a black disk on square 9, which is an X square, is vulnerable to attack from square 18. If a white disk exists on square 18, white player can then place a disk on square 0 and earn the corner square.

C squares similarly provide the opponent access to corner squares. If there is a black disk on square 48 (a C square), and a white disk on square 40, white player can then place a disk on square 56 and secure the corner location. Each of the eight C squares provides access to a corner square for the opposing player either through a vertical or horizontal attack.

On the contrary, X squares and C squares can be advantageous in situations where the adjacent corner is already occupied, usually near the endgame. For example, if there is a black disk on square 63, then having a black disk also on square 54 (an X square), or a black disk on square 55 or 62 (C squares), is not detrimental to black since black player cannot lose the already secured corner at square 63. For this reason, X squares and C squares should only be avoided if the adjacent corner is empty.

3.5 Maximum Disk Strategy

The goal of Othello is to have the most of one's own colored disks on the board at the end of the game. This winning condition gives rise to the natural strategy of attempting to maximize the number of one's own colored disks in every turn. That is, using the maximum disk strategy, a player will try to flip as many of the opponent's disks as possible on every turn. This is often the first strategy employed by beginner Othello players, as it gives them the illusion of winning in the beginning and mid-game stages of the game since the board will be littered with disks of one's own color. Unfortunately, the maximum disk strategy ends up limiting a player's mobility (see next section), which in turn usually ends up costing a player the valuable corner squares.

3.6 Maximum Mobility Strategy

The maximum mobility strategy has blossomed from the fact that the 4 corner squares on the Othello board are often critical to winning. We would like to avoid the X squares and C squares for as long as possible, so that hopefully, our opponent will be forced to play a disk onto an X square or C square, giving us potential access to a coveted corner square. Ideally, we would like to have 5-15 (or possibly more) valid moves on each turn so that we can avoid the X squares and C squares for as long as possible. In addition to maximizing our mobility, we would like to minimize our opponent's mobility. By limiting the number of valid moves our opponent has each turn, we are in effect forcing them to eventually place a disk onto an X square or C square if they have no other options.

Maximizing mobility is context dependant. Different board states and patterns lead to many methods of maximizing mobility. However, one of the simplest methods to maximize our mobility while minimizing our opponents mobility is by *minimizing* the number of disks we have on the board (in the beginning and mid-game). The fewer disks we have, the fewer options our opponent will have in flipping our disks, which will ideally eventually lead our opponent to unwillingly placing a disk on an X or C square.

3.7 Strategy Summary

The above strategies are simple observations that are meant to transition a beginner Othello player into a logical one, so that he or she may start critically thinking about how to win a game as opposed to placing disks by trial and error and seeing what happens (which in fact, can also be a decent way to learn!). In short, if you are new to Othello, a decent initial strategy can be to try to minimize the number of disks you flip each turn (to maximize your mobility) in addition to avoiding the X squares and C squares for as long as possible. This will give you the upper hand in attaining as many corner squares as possible, which will be your gateway to victory.

For more advanced strategies, see [3], which teaches additional concepts such as wedges, frontiers, tempo, parity, and stoner traps, or see [4], which is a set of video tutorials covering basic and intermediate strategies such as checkerboarding and flat theory. For a thorough strategy guide intended to turn you into an expert Othello player, see [5], written by former Othello world champion Brian Rose.

4 System Architecture

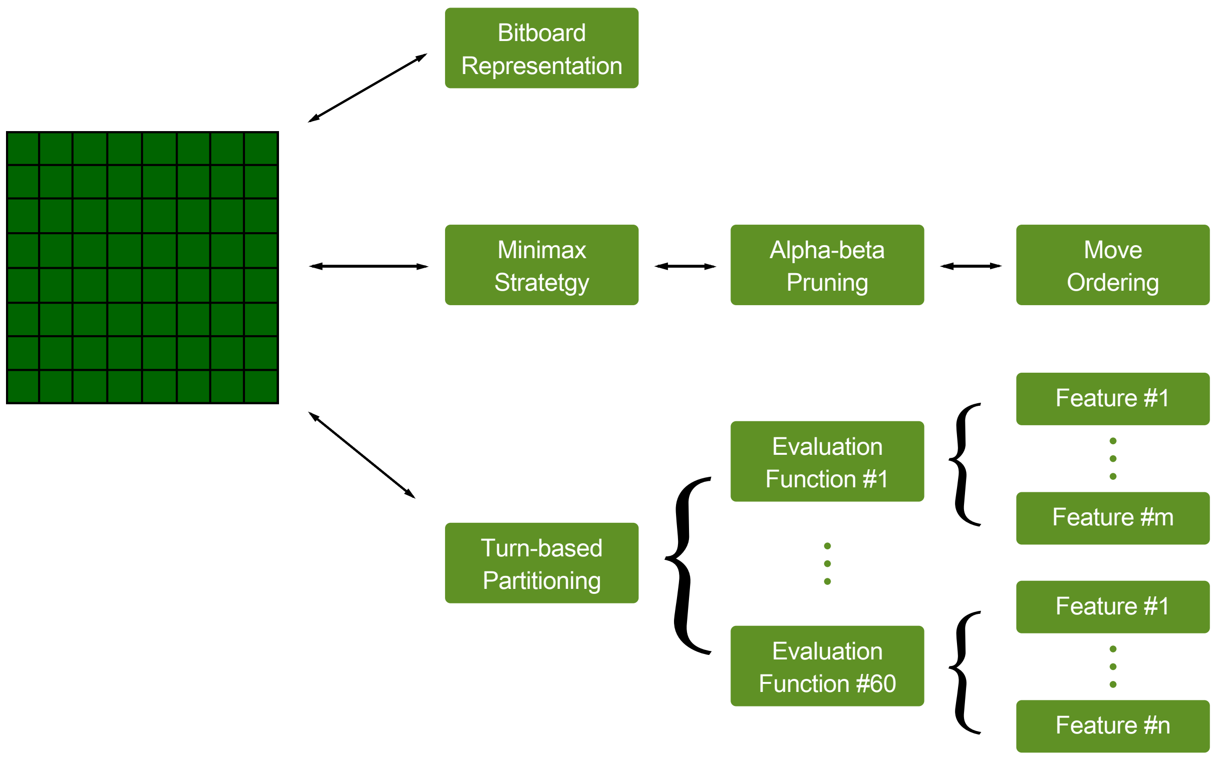


Figure 6: High-level Design

The foundation of the *tbPony* system relies on bitboards as efficient data structures. A comparable version without bitboards has also been created and is discussed in Section 5. The foundation of our search algorithm is Minimax search which is a traversal of our game tree under the assumption that both players play optimally. The theory and design of Minimax search is presented in Section 6. There are numerous improvements that can be applied to this search strategy, and Alpha-beta pruning (Section 7.1) and move order selection (Section 7.2) are two such improvements that are incorporated into the tbPony system. Our system will also use a depth cutoff and evaluation functions, as explained in Section 8, to stop search and estimate the utilities of boards.

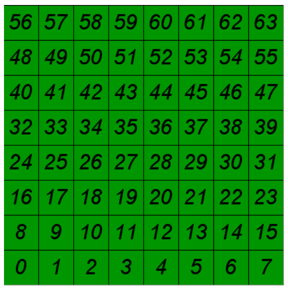
There have been a lot of game-playing strategies that have used this same foundation, but the evaluation functions used to analyze boards have been too general. The idea of turn-based partitioning is introduced in Section 9 as a way to classify boards into categories and evaluate them separately. This enables us to have a more specific evaluation function for each set of boards, resulting in more accurate estimates of utilities. By generating better estimates, we improve the strength of our computer AI. A graphic overview of the tbPony system is presented in Figure 6.

5 Bitboards

Efficient data structures are key to the strength of the computer A.I. The most important piece of information that we need to store is the state of the Othello board, so we would like to store it as efficiently as possible. During game play, we will be generating millions of additional boards for analysis. By using as few bits as necessary to represent the Othello board, we will be saving on space, but more importantly, we will be saving on processing time as new boards can be generated faster. This is where bitboards come to the rescue.

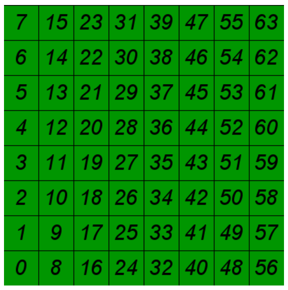
5.1 Sample Bitboard Representations

A bitboard is basically a sequence of bits used to represent a board. Each bit can represent a single piece of information. A common approach is to use 64 bits to store all of the black disks on the board. Another 64 bits can be used for the white disks, enabling us to efficiently store the locations of all the disks on the board using just 128 bits. Multiple possible numberings of the 64 squares on the board exist. The four most common numberings are shown in Figures 5. The big-endian



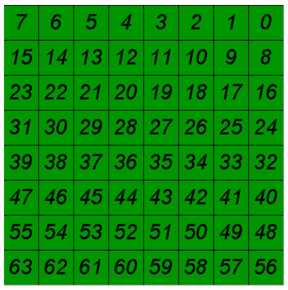
56	57	58	59	60	61	62	63
48	49	50	51	52	53	54	55
40	41	42	43	44	45	46	47
32	33	34	35	36	37	38	39
24	25	26	27	28	29	30	31
16	17	18	19	20	21	22	23
8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7

(a) Little-Endian Rank-File



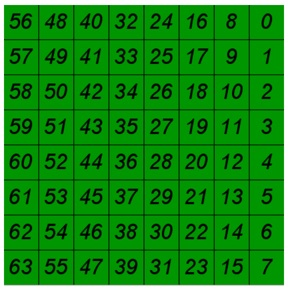
7	15	23	31	39	47	55	63
6	14	22	30	38	46	54	62
5	13	21	29	37	45	53	61
4	12	20	28	36	44	52	60
3	11	19	27	35	43	51	59
2	10	18	26	34	42	50	58
1	9	17	25	33	41	49	57
0	8	16	24	32	40	48	56

(b) Little-Endian File-Rank



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

(c) Big-Endian Rank-File



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

(d) Big-Endian File-Rank

Figure 7: Four common board numberings

numberings have the additional benefit of corresponding to the natural ordering of the digits in binary and hexadecimal numbers, since these numbers are always written in big-endian format. The big-endian rank-file numbering (Figure 7c) is the numbering we use here.

5.2 32-bit vs. 64-bit Architectures

It is important to note that most CPUs are currently 64-bit, which means that a 64-bit operation can be completed in one instruction. Coincidentally, we have 64 spaces on the board, so we were able to use 64 bits to represent all of the black disks on the board. In this scenario, a 64-bit machine grants us the ability to change any configuration of black disks on the board to any other configuration in a single instruction! On the other hand, changing the configuration of black disks on a 32-bit machine with a 64-bit representation of black disks would be less efficient since it would require at least two instructions to change 64 bits.

Nowadays, most architectures are 64-bit, so the negative effects of using a 64-bit bitboard on a 32-bit architecture is less of a concern. However, these subtleties are important to understand to be able to effectively use bitboards in variants of Othello that are played on different sized boards such as 10x10 or 16x16 boards. On such larger boards, bitboards become less efficient as 64 bits are not enough bits to represent the locations of all of a player's disks, so more instructions will be necessary for each change in disk locations.

5.3 Simple Bit Operations

In any implementation of bitboards, we need to have any easy way to "get", "set", and "clear" bits. Since our board is represented by a 64-bit long in Java, we need an easy way to get any of the 64 bits on the board.

```java
/**
 * Determines if bit exists at position represented by "tileNum"
 * @param boardRep  the board representation
 * @param tileNum   the numbered tile to change to 1
 * @return          true if bit exists at position. false otherwise
 */
public static boolean getBit(long boardRep, byte tileNum){
    return (boardRep & (1L << tileNum)) != 0;
}
```

Setting a bit should be just as easy as getting a bit. The following function will set any 0 bit on the board to 1.

```java
/**
 * Changes bit at "tileNum" to 1 in the long representation of a board
 * @param boardRep the board representation
 * @param tileNum  the numbered tile to change to 1
 * @return         the board as a long
 */
public static long setBit(long boardRep, byte tileNum){
    return boardRep | (1L << tileNum);
}
```

To clear a bit (changing a 1 bit to a 0), we simply use a mask as shown below.

```java
/**
 * Changes bit at "tileNum" to 0 in the long representation of a board
 * @param boardRep the board representation
 * @param tileNum  the numbered tile to change to 0
 * @return         the board as a long
 */
public static long clearBit(long boardRep, Point pos){
    byte tileNum = getTileNum(pos);
    long mask = ~(1L << tileNum);
    return boardRep & mask;
}
```

In addition to getting, setting, and clearing bits, we often need the ”cardinality” of a board, which represents the number of bits in the long representation of the board that are set to 1. The most straightforward way to get the cardinality is to look at each bit individually as shown below.

```java
/**
 * Counts the number of bits set in a long
 * @param num The 64-bit "long" to count the bits of
 * @return    The number of bits set in the "long" (0 to 64)
 */
public static byte cardinality(long num){
    byte result = 0;
    for (int i = 0; i < 64; i++, num >>= 1){ //assumes 64-bit long
        if ((num & 1) == 1)
            result++;
    }
    return result;
}
```

For more efficient algorithms to calculate cardinality, such as using parallelization to speed up computation, see [6]. For a player, let's say black player, we can have one long representing all the positions on the board where a black disk is placed. Using this long representation, we can calculate black player's score by simply calculating the cardinality of the long that represents black's disks.

5.4 Efficiency Of Bitboards

The following examples correspond to using the big-endian file-rank representation of a bitboard on a 64-bit processor. In this scenario, many common operations can be accomplished in a single instruction. Sample bitboards to represent certain types of squares are shown in Table 1.

Type of Squares	Square Numbers	Hexadecimal Representation
Corners	0, 7, 56, 63	8100 0000 0000 0081
Center Squares	27, 28, 35, 36	0000 0018 1800 0000
X-Squares	9, 14, 49, 54	0042 0000 0000 4200
C-Squares	1, 6, 8, 15, 48, 55, 57, 62	4281 0000 0000 8142
Right Column	0, 8, 16, 24, 32, 40, 48, 56	0101 0101 0101 0101
Left Column	7, 15, 23, 31, 39, 47, 55, 63	8080 8080 8080 8080
Top Row	0 - 7	0000 0000 0000 00FF
Bottom Row	56 - 63	FF00 0000 0000 0000

Table 1: Hexadecimal representations of squares

This is where our bitboard representation of the board becomes useful and efficient. For example, checking if there are any black disks at the 4 corner squares can be done by using the "&" (bitwise and) operator as such:

```
if ((blackPlayer.diskPositions & 0x8100000000000081L) != 0)
    corners = true;
```

Here, we directly compared blacks disks (which are represented as a long) with the hexadecimal representation of the 4 corner squares. Checking to see if black player has any disks on the right column is just as easy:

```
if ((blackPlayer.diskPositions & 0x0101010101010101L) != 0)
    rightColumn = true;
```

We can also easily clear the board of black's disks by simply setting its representation to 0:

```
blackPlayer.diskPositions = 0L;
```

5.5 Theory vs. Practice

In theory, representing an Othello board as a bitboard is likely the most efficient representation of the board. It enables us to do many common calculations, such as checking to see if black player has any disks on a certain row, in 1 operation. However, this efficiency comes at a cost. Using bitboards often makes for unreadable and not easily maintanable code. The above code samples are efficient but rather cryptic, and one must decide whether this efficiency is worth the added complexity to the code.

As an experiment for this research project, two copies of the Othello program were created, except that one version was fundamentally represented with bitboards, while the other version was represented using custom classes/objects that are familiar to Java. In comparing the speed of both programs, there was no significant difference in the bitboard version of the code to warrant the use of bitboards.

6 Minimax Strategy

6.1 Theory

Minimax search is the backbone of more complex search methods such as Alpha-beta pruning. It assumes that the opponent will play optimally, that is, it assumes the opponent will make the best possible move in every situation. This assumption us make our own decisions of which moves to perform, as we attempt to minimize the loss of the worst case outcome that can arise from a given node in a game tree. This is best illustrated with an example as shown in Figure 8.

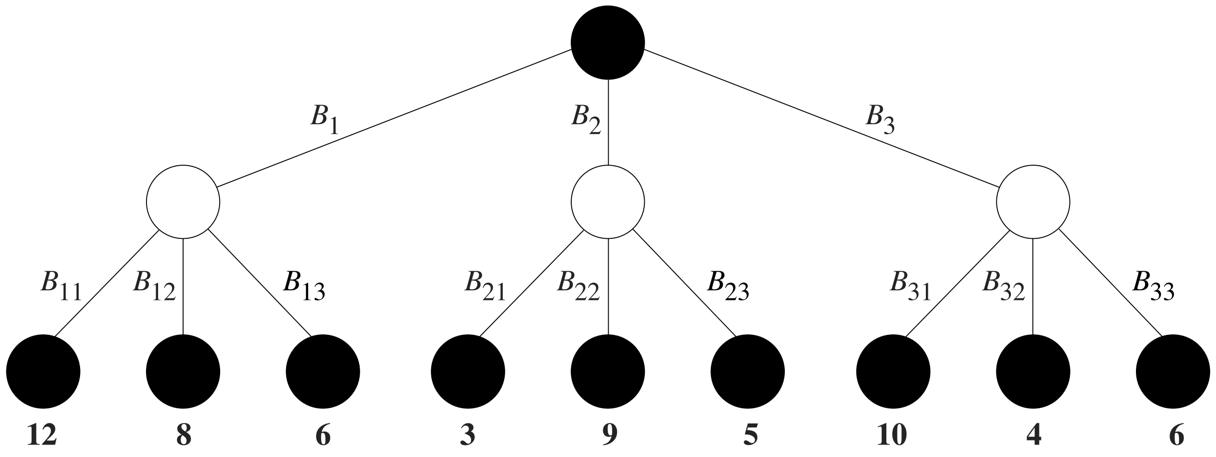


Figure 8: Minimax Example

Without loss of generality, we will assign black player to be max player, and white player to be min player. The 9 leaves of the sample game tree shown in Figure 8 show the utility that max player will receive for reaching each respective node. Max player wants to achieve the maximum utility possible and must select the branch that will achieve this maximum utility. On the other hand, min player will be trying to minimize max's utility. This is a crucial bit of information that will determine which branch max player will take. The utility of the non-leaf nodes can be calculated recursively using Equation 1.

$$\text{minimaxValue(node)} = \begin{cases} \text{Utility(node)}, & \text{if terminal(node)} \\ \max\limits_{s \in successors(node)} \text{minimaxValue(s)}, & \text{if player(node) = maxPlayer} \\ \min\limits_{s \in successors(node)} \text{minimaxValue(s)}, & \text{if player(node) = minPlayer} \end{cases} \tag{1}$$

Let's say black player takes branch B_1. In this scenario, min player has 3 options to choose from: B_{11}, B_{12}, B_{13}. Since min player's goal is to minimize max's utility, and we assumed min player is optimal, then min player will correctly choose the B_{13} branch. Now we are aware that choosing branch B_1 as max player will lead us to a final utility of 6.

What if max player chooses branch B_2? Well, in this scenario, min player will go ahead and select branch B_{21} since that is the branch with the lowest utility. This will result in a final utility of 3 for max player.

Entertaining the final option for max player, let us see where branch B_3 leads us. If max player selects branch B_3, then min player will select branch B_{32} since that leads to the node with the lowest utility. This means branch B_3 results in a utility of 4 for max player. The results are summarized in Table 2.

Max's Move	Min's Move	Resulting Utility
B_1	B_{13}	6
B_2	B_{21}	3
B_3	B_{32}	4

Table 2: Minimax results for Figure 8

As shown in Table 2, the highest utility that max player can receive is 6, so max player will select branch B_1 and min player will select branch B_{13}. The path through the game tree is highlighted in red in Figure 9.

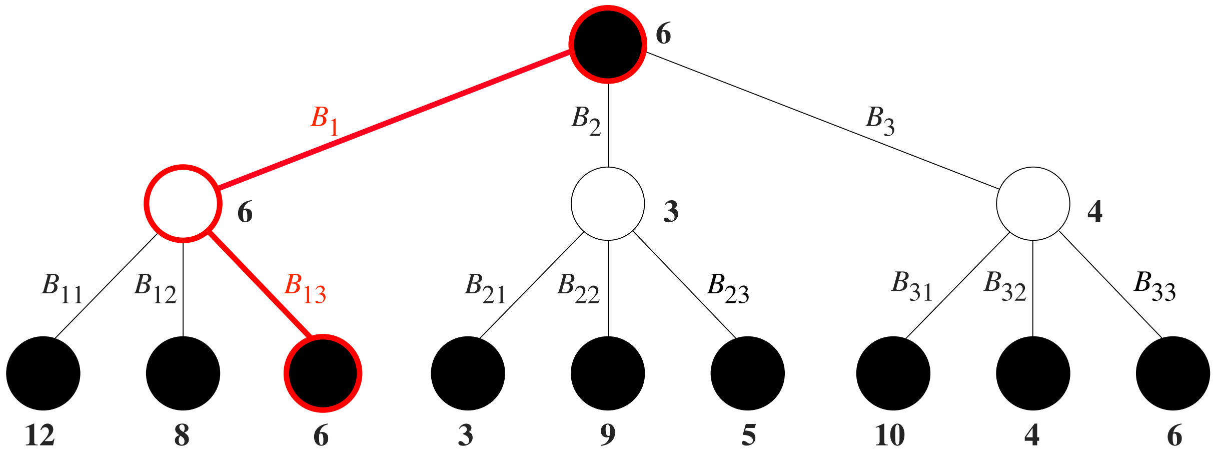


Figure 9: Minimax Example

6.2 Implementation

Once the theory of Minimax search is understood, the next hurdle is to be able to implement it. This is no trivial task, and it is recommended to refer to pseudocode that is readily available online. Included below is an implementation in Java based on Equation 1 that may also prove helpful.

```java
/**
 * Recursive function. Uses Minimax strategy to determine the next move.
 * @param board       The Board to do a "move" on.
 * @param currLevel   The current depth we've searched to in game tree (where 0
     corresponds to current state of Board).
 * @param maxDepth    The number of levels deep we should search the game tree.
 * @return            The Board after the "move" is performed.
 */
public Board minimax(Board board, int currLevel, int maxDepth){
    if (board.gameEnded || (currLevel == maxDepth))
        return board;
    Utility utility;

    ArrayList<Board> successorBoards = getAdjacentBoards(board);
    Board bestBoard = null;

    if (board.playerTurn == Color.BLACK){
        int max = Integer.MIN_VALUE;
        for (int i = 0; i < successorBoards.size(); i++){
            Board successor = successorBoards.get(i);
            Board lookaheadBoard = minimax(successor, currLevel + 1, maxDepth);
            utility = new Utility(lookaheadBoard);
22                 if (utility.value > max){
23                     max = utility.value;
24                     bestBoard = successor;
25                 }
26             }
27         }
28         else{
29             int min = Integer.MAX_VALUE;
30             for (int i = 0; i < successorBoards.size(); i++){
31                 Board successor = successorBoards.get(i);
32                 Board lookaheadBoard = minimax(successor, currLevel + 1, maxDepth);
33                 utility = new Utility(lookaheadBoard);
34                 if (utility.value < min){
35                     min = utility.value;
36                     bestBoard = successor;
37                 }
38             }
39         }
40         return bestBoard;
41     }
```

Notice the recursive nature of the function. We recursively search through *maxDepth* levels of the game tree. It is also important to note the symmetry of the outer if/else loop (lines 16-27, 28-38) as the strategies for max (black) player and min (white) player are identical except for the fact that black player is trying to maximize utilities and white player is trying to minimize utilities. There are i successor boards (line 18) that represent the i successor boards resulting from the i possible moves for the current player. For each *successor* board (line 19), we recursively search the game tree to save and return whichever board is the best result from that successor board. We save this result as *lookaheadBoard* (line 20). Whichever *successor* board resulted in the best *lookaheadBoard* will be saved as *bestBoard* (line 24) and returned (line 40). This board is the result after making the move we want to make.

6.3 Performance Against Various Opponents

It is important to note how the Minimax strategy performs against different types of opponents. We assumed our opponent plays optimally. In this scenario, our Minimax strategy gives us the optimal move to make. If we use our Minimax strategy against a non-optimal opponent, our utility can only be higher than if our opponent was optimal. The fact that our utility cannot be any lower gives us good reason to follow our Minimax strategy to get desired results regardless of how our opponent plays. Although we could have devised alternative strategies to exploit the weaknesses of our non-optimal opponent, our initial goal was to play well against an optimal opponent to minimize our worst case outcome, so the non-optimality of our opponent can be disregarded.

7 Alpha-beta Pruning

7.1 Theory

Alpha-beta pruning is an improvement on Minimax search that makes the algorithm more efficient. It relies on the premise that it is possible to compute the exact Minimax decision without expanding every node in the game tree since we can often know beforehand that certain branches cannot lead to a node with a desired utility. Let us refer back to Figure 8 to see how we can prune our tree.

If max player takes branch B_1, then branches B_{11}, B_{12}, and B_{13} must still all be explored. This results in a utility of 6 if branch B_1 is taken and there have thus far been no optimizations.

After taking branch B_1, let us consider taking branch B_2. If branch B_2 is taken, branch B_{21} will be explored next. Branch B_{21} tells us that min (white) player can select branch B_{21} to give max (black) player a utility of 3. Thus far, branch B_2 will result in a utility of less than or equal to 3. Regardless of what utilities branches B_{22} and B_{23} provide, max player will never be able to get a utility greater than 3 from branch B_2 since min player will select the lowest utility resulting from branches B_{21}, B_{22} and B_{23}. Since max player can already earn a utility of 6 by exploring branch B_1, max player can ignore branch B_2 and not spend anymore time to explore branches B_{22}, B_{23}. That is, branches B_{22} and B_{23} have been pruned.

We can use similar reasoning on branch B_3. After exploring branches B_1 and B_2, we can explore branch B_3, which leads us to explore branch B_{31}. This branch provides a utility of 10. Since $10 > 6$, where 6 was the best utility we've found so far (through branch B_1), we must further explore branch B_{32}. This branch leads to a utility of 4. We now realize that the utility from exploring branch B_3 is less than or equal to 4. Since we already have a guaranteed utility of 6 at branch B_1, we can save computation time by pruning branch B_{33} since we will not be taking branch B_3. The pruned game tree along with the optimal path are shown in Figure 10.

Notice that we arrive at the same result as when we used Minimax search. This is no coincidence, as Alpha-beta pruning still provides an optimal solution since pruning the game tree does not affect the final result.

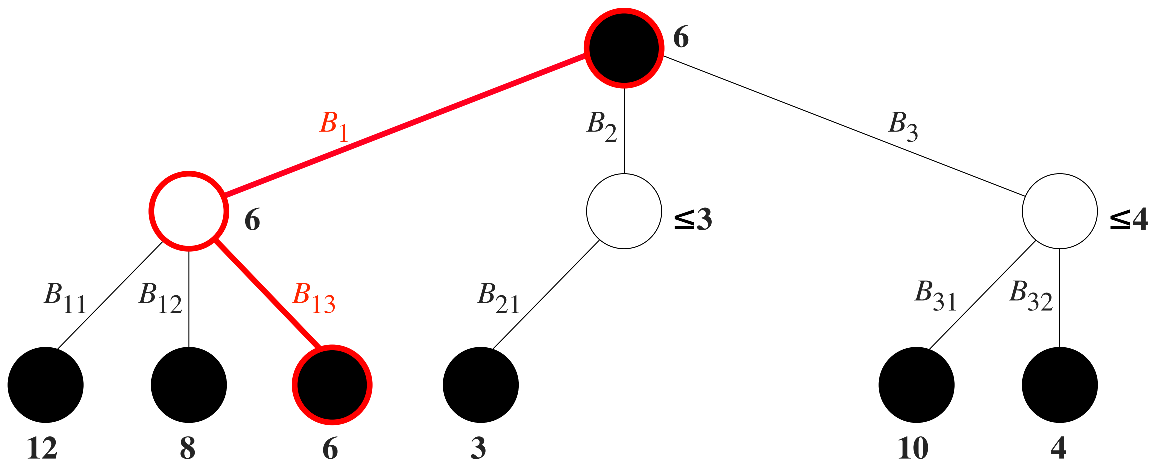


Figure 10: Alpha-beta Pruning Example

Implementing Alpha-beta pruning is very similar to implementing Minimax search. A small modification to the Minimax search code to keep track of α and β (as defined in Equation 2) is necessary so that we can save the highest and lowest utilities we have found thus far.

$$\begin{cases} \alpha = \text{the value of the highest utility found thus far.} \\ \beta = \text{the value of the lowest utility found thus far.} \end{cases} \tag{2}$$

For every node we explore while searching through the game tree, we will compare its utility to α and β to figure out if we need to prune any branches. We also update α and β with the best values we've found thus far as we are traversing the game tree.

7.2 Move Ordering

The effectiveness of Alpha-beta pruning is highly dependent on the order that we explore branches. Let's assume we have already explored branches B_1, B_{11}, B_{12}, B_{13}, and B_2 just as we did on our original Alpha-beta pruning example. Now we have a choice of 3 possible branches we can explore next: B_{21}, B_{22}, B_{23}. In our Alpha-beta pruning example, we had arbitrarily selected to take branch B_{21} first. This luckily gave us a utility that was lower than α, so we were able to immediately prune branches B_{22} and B_{23}.

If we had chosen to explore branch B_{22} first instead of branch B_{21}, we would have arrived at a utility of 9 which is *higher* than our α of 6. That means we would further have to explore B_{21} and B_{23} until either both these branches are explored, or until we arrive at a utility that is lower than α. Exploring branch B_{22} before exploring branch B_{21} would have resulted in less pruning of the game tree and more computation time to traverse our tree. This tells us that the order we select to explore the branches in our tree is important to efficiency.

For min player, we would like to try to do the best moves first. That is, we would like to explore the branches that lead to the lowest utility first. From branches B_{21}, B_{22}, and B_{23}, choosing the best branch first, B_{21}, would result in the most pruning. Similarly, for branches B_{31}, B_{32}, and B_{33}, choosing branch B_{32} first would result in the most pruning. But how do we know which moves are good? In games like Chess, we can try moves like checks or captures first, followed by moves that advance pieces forward on the board, and save moves that are less likely to be good moves, such as moving pieces backwards, for exploring last. Categorizing moves as such works well in Chess, but may be a little trickier in Othello.

If following the *Maximum Disk Strategy* (as explained in Section 3.5) we can choose whichever moves that immediately flip the most disks, and explore the trees that result from these moves first. However, the Maximum Disk Strategy was mentioned to be naive as it limits a player's mobility. A similarly simple move ordering strategy to employ would be the *Minimum Disk Strategy.* Minimizing disks would give us a rough estimate of which moves generate the most mobility (as explained in Section 3.6), and we can try those moves first instead.

As you can see, choosing which moves to try first depends on which moves *seem* like they would be good. This is somewhat subjective as we cannot immediately search the entire game tree. It is also possible to do a slightly more thorough analysis of which branches to try first. For each of the branches that can be taken from a node, we can estimate the utility of each of the immediately resulting boards using evaluation functions (see Section 8). Whichever utility is highest can be the first move to try. This method is the most versatile as any change in how we evaluate the utility of a board is immediately incorporated into our logic for selecting move orderings.

In comparison, Minimax search is a complete depth-first search of the game tree, which has a runtime of $O(b^d)$ where b is the branching factor and d is the maximum depth of the tree. Using Alpha-beta pruning with a theoretically perfect move ordering, our runtime becomes $O(b^{d/2})$. This is a huge improvement in speed as this would mean we can search a game tree with twice as much depth in the same time span.

8 Evaluation Functions

In theory, a perfect computer opponent will be able to play optimally on every turn. That is, the computer will have searched the entire game tree to decide which move is best on each turn. This is possible in simple games such as Tic-tac-toe or Checkers, but the game tree for Othello has a depth of 60 and a branching factor that is usually in the range of 5 to 15. This makes it impossible to search the entire game tree due to the sheer number of nodes. For this reason, we must stop searching after a certain number of nodes in the tree are explored. So which nodes should we explore? A common way to explore the game tree is to do depth-first search with a depth cutoff. That is, we can search a certain number of levels deep in the tree. When we get to our depth cutoff, we can use an evaluation function to estimate the utility of the board at the given node, and then again proceed searching from level 0 of the tree on a different path. This is analogous to looking d moves ahead, where d represents the depth cutoff where we use our evaluation function.

8.1 Types of Evaluation Functions

When we design evaluation functions, we must be aware of the maximum and minimum utilities that any node in tree can have. The leaf nodes in our tree correspond to when the game has ended as no further moves can be made. The utility at these leaves should be the highest and lowest utilities possible in our entire tree. In Equation 3, we select -1000 for a board where min player wins, and 1000 for a board where max player wins.

$$\text{gameEndedUtility(board)} = \begin{cases} 1000, & \text{if winner(board)} = \text{maxPlayer} \\ -1000, & \text{if winner(board)} = \text{minPlayer} \end{cases} \tag{3}$$

The tree's inner nodes should have utilities that fall in this range since for each player there is no better board that a winning board.

8.1.1 Score Differential

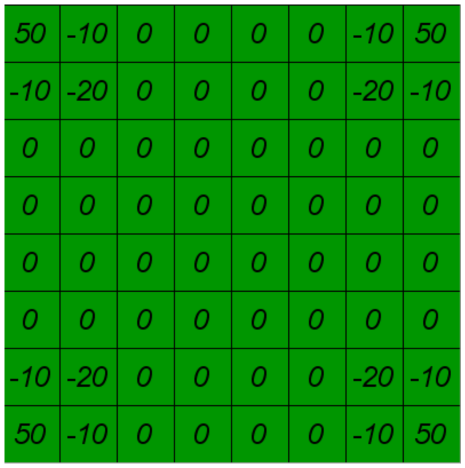
A simple way to evaluate the utility of the board is to compare the number of disks that each player has as shown in Equations 3 and 4. This method is efficient as we usually already know how many disks each player has, and a simple subtraction can quickly give us the utility of the board.

$$\text{utility(board)} = \begin{cases} \text{gameEndedUtility(board)}, & \text{if gameEnded(board)} \\ \text{numBlackDisks(board) - numWhiteDisks(board)}, & \text{otherwise} \end{cases} \tag{4}$$

Although the goal is to have more disks than the opponent at the end of the game, having more disks in the beginning or middle of the game is not a good indicator of which player is winning. To get a more accurate representation of the utility of a board we will have to design more complex evaluation functions.

8.1.2 Weighted Disk-Squares tables

Using the score differential from Section 8.1.1 as the utility of a board has a crucial flaw. It treats all disk positions equally. The positioning of disks can give us insight into who is winning. We know from Section 3.3 that corners squares are more powerful than other squares, and we know from Section 3.4 that X squares and C squares are usually weaker than other squares. When we count the number of disks each player has, we can also take into account whether any of the disks are on an X square, C square, or corner. A straightforward way to do this is to use a disk-squares table as shown in Figure 11. Here, we have assigned a utility to each square. The corners are assigned a utility of 50. This is because we want a very high reward for capturing a corner square, and we want to primarily base our strategy on capturing these squares. We have assigned a utility of -10 for C squares and -20 for X since those squares provide the opponent an advantage to gaining a corner square. For all other squares, we assigned a utility of 0 since none of the other squares are especially important. The utility for max player will simply be the sum of all the utilities on the board as shown in Equation 5.



50	-10	0	0	0	0	-10	50
-10	-20	0	0	0	0	-20	-10
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
-10	-20	0	0	0	0	-20	-10
50	-10	0	0	0	0	-10	50

Figure 11: Sample Disk-Squares Table

$$\text{utility(board)} = \begin{cases} \text{gameEndedUtility(board)}, & \text{if gameEnded(board)} \\ \sum_{n=0}^{63} \text{utilitySquare(n, board)}, & \text{otherwise} \end{cases} \tag{5}$$

Compared to our evaluation function from Section 8.1.1, we now reward players for gaining a positional advantage instead of for just flipping as many disks as possible. An improvement on this method would be to use a different disk-squares table for each stage of the game, as we will explore in Section 9.

8.1.3 Mobility-Based Evaluation

One of the best strategies widely used by experts is the maximum mobility strategy as explained in Section 3.6. We can design an evaluation function that matches this strategy. For a given board, we can calculate the number of moves each player has and, just like in Section 8.1.1, do a simple subtraction to get the utility of the board. This can be done using Equation 6

$$\text{utility(board)} = \begin{cases} \text{gameEndedUtility(board)}, & \text{if gameEnded(board)} \\ \text{mobilityMaxPlayer(board) - mobilityMinPlayer(board)}, & \text{otherwise} \end{cases} \tag{6}$$

8.2 Combining Functions

Each of the evaluation functions in Section 8.1 has its own advantages and disadvantages. None of them alone would suffice as a good strategy. We can't rely on mobility alone to win a game, just as we cannot rely on trying to maximize disks to win a game. Generally, we use a weighted sum to combine multiple evaluation features into a single evaluation function using Equation 7

$$\text{utility(board)} = \begin{cases} \text{gameEndedUtility(board)}, & \text{if gameEnded(board)} \\ \sum\limits_{i=1}^{n} w_i e_i, & \text{otherwise} \end{cases} \tag{7}$$

Here, we have linearly combined n evaluation functions, $e_1 ... e_n$, into a single evaluation function using weights $w_1 ... w_n$ to indicate which evaluation functions are most important. Non-linear combinations of evaluation functions are also possible for more precise results.

The astute reader will realize that creating and combining evaluation functions is very subjective. Determining which evaluation functions to use usually requires a deep knowledge of Othello, and human generated evaluated functions are limited by the skill of the human Othello player. The evaluation functions presented here are just a few features that can be linearly combined to evaluate a board. Often, hundreds or thousands of features are combined for more precise results. However, it is important to keep in mind that more complex evaluation functions take more computation time, so a balance needs to be compromised for time spent evaluating specific nodes and time spent exploring new nodes in the game tree.

9 Turn-based Partitioning

You may have noticed that Othello is a game of just 60 moves. Each player has a maximum of 30 decisions to make, as there are a finite number of disks and available squares on the board. This gameplay mechanic ensures that we have a 61 level game tree, where level 0 corresponds to the beginning of the game with 4 disks on the board, and level 60 corresponds to the end of the game with all 64 disks on the board. Another property characteristic to Othello is that disks can never be removed from a board. This ensures that each board on level d of our game tree will have exactly $d + 4$ disks. Alpha-beta search always compares boards that are on the same level of the tree. So for every comparison, both boards will have exactly the same number of disks (This property is not necessarily true in other games. In Chess, two boards on the same level of the search tree may have a different number of pieces).

In Section 8.2, we designed a single evaluation function (as shown in Equation 7) that can give us a utility for any of the roughly 10^{28} unique and valid Othello boards that can arise from gameplay. Although this function has many variables to analyze the game state, it is still a single function that attempts to evaluate all 10^{28} boards. What if we could somehow partition the set of possible boards into sets of similar boards, and then assign a different evaluation function to each partition? Each of our evaluation functions would then have fewer boards to analyze, and we could tailor each function more specifically to boards in each partition.

The special form of our Othello game tree gives us a natural way to partition the set of boards. We can partition our set of boards into exactly 60 different partitions: Boards with i disks on them where $5 \leq i \leq 64$. This is equivalent to partitioning the boards depending on which of the 60 turns it is in the game. Yet another way to view this partitioning is that we are partitioning by level on our 61 level game tree, where we ignore the root node at level 0.

Since our game is now partitioned into 60 time steps, we will create a different evaluation function corresponding to each time step. Now we can more easily take into account the importance of certain squares depending on whether they are captured early or late in the game. For example, in Section 8.1.2, we added a negative utility for having a disk on an X square. Near the beginning of the game, this would give the opponent a huge advantage in capturing the X square's neighboring corner square. However, placing a disk on an X square on turn 60 (the last move of the game) cannot possibly have the vulnerability of losing a corner to an opponent since the game is already over, and a negative utility in this situation makes no sense. Partitioning by turn solves this problem. If using Equation 5, we can make a more accurate disk-squares table that varies as a function of time, t, where $1 \leq t \leq 60$. Now the utilities of important squares like corners and X squares will depend on the value of t and will vary as t increases.

9.1 Static and Dynamic Utilities

Let us try to combine Equation 5 with Figure 11 into a new equation to describe the utility of a board. We will also incorporate mobility and number of disks into this equation. The resulting equation is displayed as Equation 8.

$$\text{utility(board, t)} = \begin{cases} w_1 e_1 + w_2 e_2 + w_3 e_3 + w_4 e_4, & \text{if } 1 \le t \le 59 \\ \text{gameEndedUtility(board)}, & \text{if t} = 60 \end{cases} \tag{8}$$

The values of variables e_1, e_2, e_3,and e_4 are given in Equation 9.

$$\begin{cases} e_1 = \text{number of corner squares owned} \\ e_2 = \text{number of X squares owned} \\ e_3 = \text{number of C squares owned} \\ e_4 = \text{number of disks owned} \end{cases} \tag{9}$$

For w_1, w_2, w_3, and w_4 we provide a set of static values and a set of dynamic values in Table 3. The static values correspond to using a static strategy without adding turn-based partitioning. The dynamic values are the result of creating weights as a function of time which is a convenient way to incorporate turn-based partitioning.

Variable	Static Value	Dynamic Value
w_1	50	$\begin{cases} 70 - t, & \text{if } 1 \le t \le 41 \\ 1, & \text{if } t \ge 42 \end{cases}$
w_2	-20	$\begin{cases} -50 + t, & \text{if } 1 \le t \le 41 \\ 1, & \text{if } t \ge 42 \end{cases}$
w_3	-10	$\begin{cases} -45 + t, & \text{if } 1 \le t \le 41 \\ 1, & \text{if } t \ge 42 \end{cases}$
w_4	0	$\begin{cases} 0, & \text{if } 1 \le t \le 41 \\ 1, & \text{if } t \ge 42 \end{cases}$

Table 3: Static and Dynamic Variable Values

9.2 Results: Dynamic Strategy vs. Static Strategy

To compare our strategies, we will simulate games between 2 players where each player uses a different strategy. We will be using the 3 strategies listed below.

1. **Static Alpha-beta *D*** - Alpha-beta search with a search depth of *D*. Uses Equation 8 with static values as an evaluation function.
2. **Dynamic Alpha-beta *D*** - Alpha-beta search with a search depth of *D*. Uses Equation 8 with dynamic values as an evaluation function.

3. **Random** - Moves are chosen randomly.

Strategy	Opponent Strategy	# of Games	# of Wins	# of Losses	% Won
Static Alpha-beta 1	Random	4000	3075	925	76.875%
Static Alpha-beta 2	Random	4000	3348	652	83.700%
Static Alpha-beta 3	Random	4000	3411	589	85.275%
Static Alpha-beta 4	Random	4000	3376	624	84.400%
Static Alpha-beta 5	Random	4000	3348	652	83.700%
Static Alpha-beta 6	Random	4000	3378	622	84.450%
Totals					
Static Alpha-beta	Random	24000	19936	4064	83.067%

Table 4: Results - Static Alpha-beta vs. Random

Strategy	Opponent Strategy	# of Games	# of Wins	# of Losses	% Won
Dynamic Alpha-beta 1	Random	4000	3679	321	91.975%
Dynamic Alpha-beta 2	Random	4000	3815	185	95.375%
Dynamic Alpha-beta 3	Random	4000	3778	222	94.450%
Dynamic Alpha-beta 4	Random	4000	3777	223	94.425%
Dynamic Alpha-beta 5	Random	4000	3812	188	95.300%
Dynamic Alpha-beta 6	Random	4000	3792	208	94.800%
Totals					
Dynamic Alpha-beta	Random	24000	22653	1347	94.388%

Table 5: Results - Dynamic Alpha-beta vs. Random

9.3 Analysis of Results

For the Static Alpha-beta vs. Random matchups, our simple disk-squares table with static values performed moderately well against our novice opponent. Table 4 shows a win percentage of 83.067%. This shows that solely using a disk-squares table with static values is a mediocre strategy.

For the Dynamic Alpha-beta vs. Random matchups, Table 5 shows a win percentage of 94.388%. This is a decent improvement on our simple strategy due to the fact that each square on the board can now have 60 different values corresponding to the 60 turns in the game. This gave us the flexibility to vary the amount of utility each square provides depending on how far we have progressed in the game. For example, in Table 3 we rewarded a higher utility for earning a corner square in the beginning of the game and a moderate utility for earning a corner square in the middle or end of the game.

Turn-based partitioning helped make our evaluation function dynamic by being dependent on time. We can apply this theory to more elaborate evaluation functions to make them dynamic. However, choosing good values for the weights in Table 3 is a challenging task that is limited by the strength of the human player designing the values. Further improvements can be made by altering these values in an attempt to improve win percentages. If designed correctly, using turn-based partitioning should provide better results than using static values as weights.

9.4 Improving Our Evaluation Function

Ideally, we want a win percentage much closer to 100% since intelligent players should rarely lose against novices that move randomly. The evaluation function we used in Equation 5 was purposely simple in an effort to illustrate a point. Adding turn-based partitioning to this equation gives us more accurate results, but this equation only provides us with a simple evaluation function based on disk-squares tables. We can use the more precise Equation 7 to incorporate other variables such as player mobility into our evaluation function. Further adding turn-based partitioning to Equation 7 would give us the most accurate utilities for our boards thus far. It is also important to note that for the last dozen or so moves of the game, we can actually search the entire game tree to find the optimal solution in a practical amount of time, which is approximately a minute on a current laptop for turns 48 and 49, and a few seconds for turns 50 to 60 to find the optimal solution. Let us define our new and improved evaluation function as Equation 10.

$$\text{utility(board, t)} = \begin{cases} w_1e_1 + w_2e_2 + w_3e_3 + w_4e_4 + w_5e_5, & \text{if } 1 \leq t \leq 59 \\ \text{gameEndedUtility(board)}, & \text{if t} = 60 \end{cases} \tag{10}$$

The values of variables $e_1, ...e_5$ and w_5 are given in Equation 11. Let us also define a *bad* X square as owning an X square that is adjacent to an unoccupied corner. Such squares are bad in the sense that they give the opponent easy access to placing a disk on the adjacent corner. Similarly, a bad C square is an occupied C square that is next to an unoccupied corner. By considering two or more squares simultaneously, we have grouped these individual features into patterns we can detect. For more examples of combining squares into patterns, such as 2x4 corner patterns or edge patterns, see [7].

$$\begin{cases} e_1 = \text{number of corner squares owned} \\ e_2 = \text{number of } \textit{bad} \text{ X squares owned} \\ e_3 = \text{number of } \textit{bad} \text{ C squares owned} \\ e_4 = \text{number of disks owned} \\ e_5 = \text{number of possible valid moves} \\ w_5 = 1 \end{cases} \tag{11}$$

Variable	Dynamic Value
w_1	$66 - t$
w_2	$-60 + t$
w_3	$-63 + t$
w_4	$\begin{cases} 0, & \text{if } 1 \leq t \leq 43 \\ 1, & \text{if } t \geq 44 \end{cases}$

Table 6: Dynamic Variable Values

In addition to using Equation 10 to evaluate utilities of board, we will do a complete search of the game tree for turns 50 to 60. Let us call our new strategy: **Final Alpha-Beta**. Using the variable values from Table 6, we arrive at the results shown in Table 7.

Strategy	Opponent Strategy	# of Games	# of Wins	# of Losses	% Won
Final Alpha-beta 1	Random	16000	15810	190	98.813%
Final Alpha-beta 2	Random	16000	15927	73	99.544%
Final Alpha-beta 3	Random	16000	15890	110	99.313%
Final Alpha-beta 4	Random	16000	15893	107	99.331%
Final Alpha-beta 5	Random	16000	15892	108	99.325%
Final Alpha-beta 6	Random	16000	15891	109	99.319%
Totals					
Final Alpha-beta	Random	96000	95303	697	99.274%

Table 7: Results - Final Alpha-beta vs. Random

Our win percentage of 99.274% is decent against a novice opponent. We will explore how to improve this win percentage further in Section 10.

10 Further Improvements

Numerous methods exist for improving various aspects of our game tree search. Minimax search may be improved by using parallel processing [8] to simultaneously traverse multiple branches of our game tree. The Minimax search algorithm we used also employed a depth cutoff which leads to the horizon effect. It may be the case where there is an excellent move that we just missed (such as gaining a corner square) that we would have seen if we had searched just one move deeper in our game tree. By employing quiescence search, we further evaluate noteworthy positions (such as positions where gaining or losing corner are likely) by searching at a higher depth for such unstable positions [9].

Designing evaluation functions can also be improved by finding professional Othello players to design these functions. Differential evolution may be used in this case to leverage the collective intelligence of a set of expert Othello players [10]. A collection of Othello games played between human experts is also available online and can be used to increase the intelligence of our AI. Many patterns exist in the beginning stages of an Othello game and are known as openings. By analyzing previous games, we can learn which openings lead to favorable positions and use this knowledge to aid us in the first several moves of the game [11].

10.1 Alternative Methods

10.1.1 Monte Carlo Search

Evaluation functions are just one method of guiding our AI's intelligence. Another feasible alternative is Monte Carlo tree search, which has successfully been applied to Othello [12]. Monte Carlo search is used to estimate the utility at a node. To determine this utility, games are simulated from this game state by having both players make random moves. The outcome of the game is then recorded as a data point. Running thousands of trials from a node gives us an estimate of the likelihood of winning from a given board state. These trials are independent of each other, enabling us to leverage parallel processing on the GPU for faster computations [13].

10.1.2 Neural Networks

Neural networks have been used in Chess as early as in 1994 to represent Chess endgames [14]. They have also been used to determine the positional value of various pieces [15] and to discover powerful moves [16]. When designing evaluation functions in Othello, we were severely limited by the strength of the human Othello player that designed these functions. Neural networks bypass having expert Othello knowledge as a necessity to create accurate evaluation functions. Instead, these evaluation functions can be learned through the evolution of neural networks through game simulations [17].

11 Conclusion

In this paper, the components of building an intelligent Othello system have been outlined. The principles of game tree search guided us into choosing which moves to make and were used as the foundation of our computer AI. Various optimizations, both in implementation and in the algorithm creation process, have been presented and discussed in detail. The idea of turn-based partitioning is another such optimization that applies to evaluation functions and has been demonstrated as a useful tool for more accurate utility estimates. This partitioning was applied to a simple evaluation function and showed an immediate improvement in win percentages. Further research can be done in applying turn-based partitioning to more complex evaluation functions, possibly ones with hundreds or thousands of features, to see if such partitioning also results in higher win percentages for more complex functions. Regardless of whether evaluation functions are generated by hand by humans or through reinforcement learning by computers, turn-based partitioning remains a useful tool to generate a set of more precise evaluation functions to replace a single encompassing evaluation function.

11.1 Experiment With The Code

The accompanying code is written in Java and is available publicly on Github. This includes a 1-player version with a GUI to test your Othello skills against an artifical opponent, and a computer vs. computer (GUI-less) version for generating simulations using different search strategies.

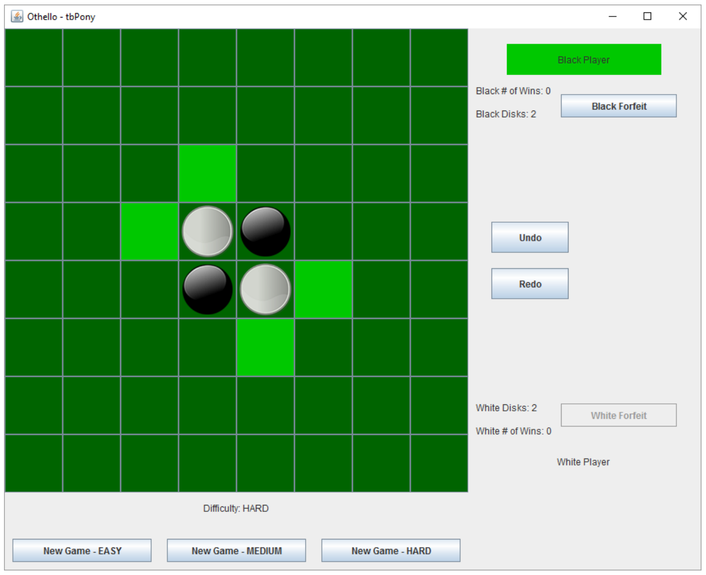


Figure 12: Playable Othello Game

Acknowledgements

I would like to thank my thesis advisor Dr. ChengXiang Zhai for his availability and expert guidance, which greatly aided the current work. I would also like to thank Sunaya Shivakumar for proofreading and designing figures for this paper.

References

[1] S. Hedberg. Smart games: beyond the deep blue horizon. *IEEE Expert*, 12(4):15–18, Jul 1997.

[2] F. Y. Wang, J. J. Zhang, X. Zheng, X. Wang, Y. Yuan, X. Dai, J. Zhang, and L. Yang. Where does alphago go: from church-turing thesis to alphago thesis and beyond. *IEEE/CAA Journal of Automatica Sinica*, 3(2):113–120, April 2016.

[3] Emmanuel Lazard. Strategy guide. http://radagast.se/othello/Help/strategy.html. Accessed: 2016-07-02.

[4] David. Othellolessons. https://www.youtube.com/user/Othellolessons/videos. Accessed: 2016-07-02.

[5] Brian Rose. A minute to learn... a lifetime to master. 2005.

[6] chessprogramming. Population count. https://chessprogramming.wikispaces.com/Population+Count. Accessed: 2016-08-12.

[7] Michael Buro. An evaluation function for othello based on statistics, 1997.

[8] D. G. Kirkpatrick and T. Przytycka. An optimal parallel minimax tree algorithm. In *Parallel and Distributed Processing, 1990. Proceedings of the Second IEEE Symposium on*, pages 293–300, Dec 1990.

[9] H. Kaindl and A. Scheucher. Reasons for the effects of bounded look-ahead search. *IEEE Transactions on Systems, Man, and Cybernetics*, 22(5):992–1007, Sep 1992.

[10] T. Takahama and S. Sakai. Emerging collective intelligence in othello players evolved by differential evolution. In *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 214–221, Aug 2015.

[11] Kyung-Joong Kim and Sung-Bae Cho. Evolutionary othello players boosted by opening knowledge. In *2006 IEEE International Conference on Evolutionary Computation*, pages 984–991, 2006.

[12] P. Hingston and M. Masek. Experiments with monte carlo othello. In *2007 IEEE Congress on Evolutionary Computation*, pages 4059–4064, Sept 2007.

[13] K. Rocki and R. Suda. Large-scale parallel monte carlo tree search on gpu. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 2034–2037, May 2011.

[14] C. Posthoff, S. Schawelski, and M. Schlosser. Neural network learning in a chess endgame. In *Neural Networks, 1994. IEEE World Congress on Computational Intelligence., 1994 IEEE International Conference on*, volume 5, pages 3420–3425 vol.5, Jun 1994.

[15] E. Vzquez-Fernndez, C. A. Coello Coello, and F. D. Sagols Troncoso. Assessing the positional values of chess pieces by tuning neural networks' weights with an evolutionary algorithm. In *World Automation Congress (WAC), 2012*, pages 1–6, June 2012.

[16] C. Dendek and J. Mandziuk. A neural network classifier of chess moves. In *Hybrid Intelligent Systems, 2008. HIS '08. Eighth International Conference on*, pages 338–343, Sept 2008.

[17] S. Y. Chong, M. K. Tan, and J. D. White. Observing the evolution of neural networks learning to play the game of othello. *IEEE Transactions on Evolutionary Computation*, 9(3):240–251, June 2005.