NEA Documentation

Hugh O'Donnell

Candidate Number: 4102
Qualification Code: 7517D

Candidate Number: 4102
Qualification Code: 7517D Centre Number: 61679

Contents

Chapter 1

Analysis

1.1 My Project

Othello is an abstract strategy game played on an eight by eight board. I will implement a digital version of Othello with a graphical user interface and a computer player. The end user will be Eugene O'Donnell.

1.2 Othello

1.2.1 Rules

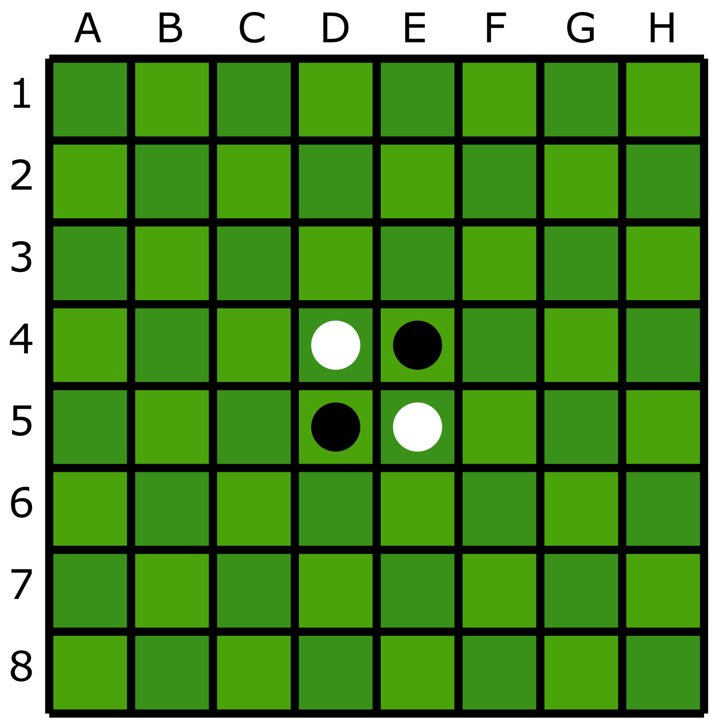


Figure 1.1: An Othello board's starting position. By convention, the black piece closest to each player is on that player's left, although this is inconsequential due to symmetry.

Othello is an abstract strategy game. Two players, black and white, play against each other on an eight by eight board placing one disc on a square each turn. Players alternate turns with black moving first. If a player has no legal moves when it should be their turn, it is the other player's turn again. If both players have no legal moves, the game ends, and the winner, if they exist, is the player with the most discs of their color on the board. If the winner does not exist, i.e., when there are the same number of black discs and white discs on the board, the game ends in a draw.

When a player makes a move, they place a disc on the board with their color facing upwards. If there is a continuous line of discs of the opposing color between the disc just placed and another disc of the player's color, those discs of the opposing color are captured (flipped), and thus changed to the color of the player who made the move. If, and only if, a move causes discs to be flipped, that move is legal. The official rules are here: `https://www.worldothello.org/about/about-othello/othello-rules/official-rules/english`.

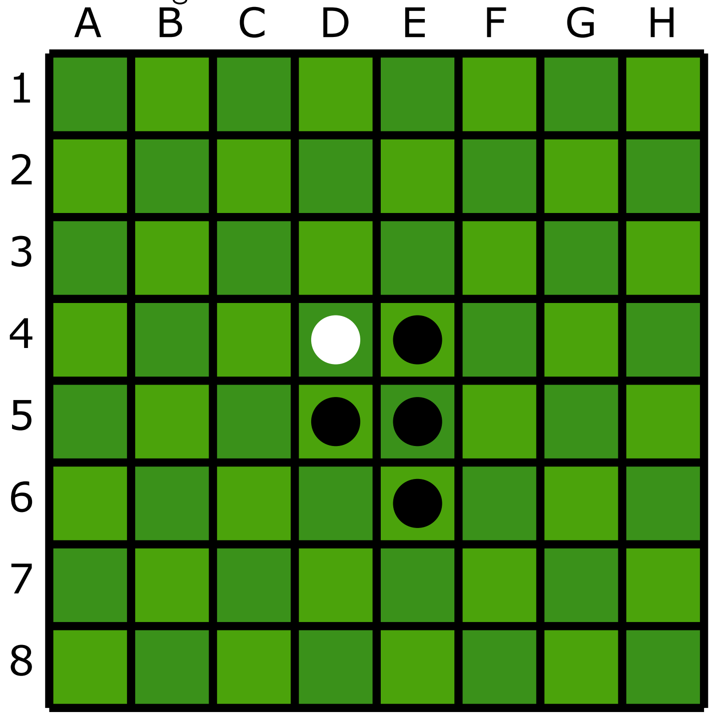


Figure 1.2: The position in 1.1 if black plays E6

1.3 Computer Othello

Almost all algorithms for estimating a good move for an abstract strategy game such as Othello involve an inexhaustive search of the game tree. The game tree consists of nodes which represent different Othello positions, and edges from parent nodes to child nodes represent the fact that the position which the child node represents can be reached in exactly one move from the position represented by the parent node. An evaluation function, i.e., a function of a position which evaluates how advantageous that position is to a particular player, is common to many, but not all, game-playing algorithms.

1.3.1 Minimax

Minimax is a brute force algorithm for searching the game tree, parameterized by an evaluation function, f and a search depth, n. All of the possible positions after n moves will be evaluated with f. The move which *guarantees* the most advantageous position after n moves (the one with the greatest evaluation) is chosen. Minimax at depth n has time complexity $\mathcal{O}(b^n)$ and space complexity $\mathcal{O}(bn)$, where b is the branching factor, i.e., the average number of legal moves in an Othello position, which for Othello is about 10. Many of the other algorithms for finding a good move are similar to minimax. I will use minimax with a variation of alpha-beta pruning in order to find the best move given some particular Othello position. Alpha-beta pruning is purely an optimization of minimax; it does not change the result. As minimax is better at greater depths, the depth can serve as a difficulty which the end user can adjust.

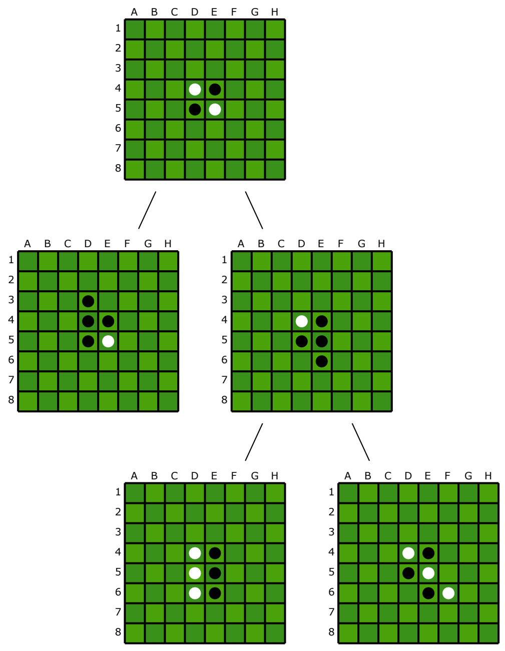


Figure 1.3: A small subgraph of Othello's game tree. The size of the whole tree is estimated at 10^{54} nodes.

1.3.2 Evalutaion Functions

The evaluation function is crucial to the strength of the computer player; it is far more important than the search algorithm. There are several ways to evaluate an Othello position. I will discuss two below.

Disc Square Tables

A disc-square table assigns some value to each square on an Othello board, with squares with higher values being better squares to have a piece on. The value of a position can be calculated by subtracting the sum of the squares which white occupies from the sum of those which black occupies. The table can be changed between stages of the game because the most important squares to occupy change over the course of the game. This method of evaluation is rather volatile because many discs can be flipped from move to move so the evaluation will change quickly.

Mobility-Based Evaluation

This evaluation function gives preference to positions with more available moves and fewer frontier squares, i.e., empty squares adjacent to a player's discs. In Othello, there is no stalemate; if a player has no legal moves, they forfeit their turn, which is unfavourable and something which this evaluation function optimizes for. Mobility-based evaluation is generally better than disc-square tables because the mobility of a player's position changes less from turn to turn than the score from a disc-square table, as rows of discs can be flipped which drastically changes the disc-square table evaluation. The number of available moves and frontier squares can be computed quickly using bitboards, which are discussed in 2.2.1.

My Choice of Evaluation Function

I will use a combination of both disc square tables and mobility based evaluation. The disc square table I use will only value occupancy of the corners. It is impossible to flip a disc at one of the corners so this method does not succumb the aforementioned shortcoming of a disc-square table. Everything else in my evaluation function will be mobility-based.

1.4 End User and Requirements

The end user is Eugene O'Donnell, my father. He wants a digital implementation of Othello because it is easier to play against a computer than to find a human to play against and carry the board and pieces around. It is also easier to find a good match with a digital version of Othello because the difficulty of the computer player can be changed easily.

1.4.1 Interview with Eugene O'Donnell

Me: Can you tell me about the project you're looking to have developed?

Eugene: Yes, I want a digital implementation of the game Othello with a computer player that has a graphical user interface, and a variety of selectable options for the user.

Me: Can you tell me more about what specific features you would like to see in the graphical user interface?

Eugene: I want the user to be able to choose the difficulty of the computer player, whether they want to play white or black, and the color of the disc and squares on the board with a color picker. And, when a square is clicked, a disc should be placed if it's the human player's turn and the move is legal.

Me: Are there any other specific requirements for the computer player?

Eugene: Yes, the computer player should make moves that are good enough to beat the user in a reasonable length of time.

Me: OK. Are there any other features you would like to see in the game?

Eugene: Yes, I want the graphical user interface to display the number of discs belonging to each player on the board, and when the game is over, it should show who won, or if there was a draw. I want a button which changes the size of the discs on the board. I would also like the legal moves to be shown on the board; I'm a little rusty on the rules.

Me: Is there anything else you would like to add?

Eugene: No, that's all for now.

End of Interview

1.4.2 Objectives

Here are my objectives, based on the end user's requirements:

1. Upon launching the program, a graphical user interface should be shown to the user.
2. When playing Othello, a disc should be placed on the board when a square is clicked if and only if it is the human player's turn, and the move which they are making is legal.
3. The difficulty of the computer player must be selectable through the graphical user interface.

4. Whether the human plays white or black must be selectable through the graphical user interface.
5. The user must be able to choose the color of the discs and squares on the board with a color picker.
6. The computer player must make moves good enough to beat the end user, taking no longer than five seconds to make one move.
7. The graphical user interface should display the number of discs belonging to each player on the board.
8. When the game is over, the graphical user interface should show whether the game was won or drawn, and if the game was won, which player won.
9. The graphical user interface must remain responsive while the computer player is computing a move.
10. The user must be able to toggle between small and large discs when playing Othello.
11. The legal moves of the player whose move it is must be shown by the graphical user interface on the board.

1.5 Proposed Solutions

Here are my evaluations and decisions over how I will achieve the objectives.

1.5.1 Programming Language

I know Python and C++ well enough for this project. I am also considering using Rust, although my knowledge of it is limited, because it is memory safe and about as fast as C++. I know I will have to use multi-threading in some capacity in order to have the GUI (Graphical User Interface) respond while the minimax is running. I want the minimax to run as fast as possible in order to search as far ahead as possible within a reasonable amount of time.

Advantages and Disadvantages

Python	C++	Rust
+ Memory safe (garbage collected). + I have experience. + Interpreted, so more portable than Rust or Python. - Interpreted, so executes relatively slowly.	+ Fast. + I have experience. + Statically typed. - Code must be compiled each time, which can take a long time. - Not memory safe, so bugs are more easily created.	+ Memory safe (borrow checker), in fact data races are impossible. + Fast. + Statically typed. - Code must be compiled each time, and the compilation is slower than in C++. - I don't have much experience. - There are fewer libraries available than in Python or C++

I have decided to go with Rust, because I want to learn it and the impossibility of data races eliminates a whole class of bugs which could be introduced in the implementation of multithreading.

1.5.2 Graphical User Interface Library

I decided to use FLTK because there are not too many GUI libraries for Rust, and this one seemed to be well documented and well used, as it was originally a C++ library. I do not know that much about all the other GUI libraries because I have not tried many of them, but FLTK works well enough for me so I will use it. All of the objectives, except objective six, will be at least partially dependent on the graphical user interface.

1.5.3 Keeping the Graphical User Interface Responsive

If all the computation was done on a single thread, when the computer player was finding a move using minimax, the user would have to wait until the computer player had finished computing a move before any of the buttons of the graphical user interface would work. In order to fulfill objective nine, I must use multi-threading to allow the computer player to find a move in one thread while another thread keeps checking for interactions with the graphical user interface from the user. This will mean that the thread checking for interactions with the user will never stop to compute a move, so the graphical user interface will remain responsive regardless of whether it is the computer player's turn.

1.6 Work Plan

I have fifteen weeks to finish coding the project. This is how I plan to complete the whole project within the allowed time.

Weeks 1-4	Weeks 5-8	Weeks 9-12	Week 13
Write the digital representation of Othello without worrying about any user interface. Implement functions for making moves, evaluating positions, searching the game tree, generating child positions from a single position, and the computer player.	Write a command line interface allowing a human to play against the computer and to test different evaluation functions by making the computer play against itself. This should make any bugs in any of what was written in weeks 1-4 apparent.	Write the graphical user interface allowing a human to play against the computer by clicking on the board. Complete all the objectives.	Test every part of the graphical user interface for bugs by playing against the computer at different depths, clicking all the buttons available, etc.

I will check regularly with my end user that the project is going in the right direction, and that everything which they have requested is being implemented in the way they envisaged.

Chapter 2

Documented Design

2.1 Software Architecture

There are two main responsibilities of my software; I need to implement a digital representation of an Othello board with a strong computer player, and I need to implement a GUI which will allow my client to play against the computer. The digital representation of Othello should know nothing of the GUI, but the code for the GUI will have to know how the board is represented, how a move is represented and so on in order to draw the board and allow the player to make moves. For this reason, I will write the digital representation of Othello before writing the GUI because the GUI depends on the digital representation of Othello, but the digital representation is independent of the GUI. Figure 2.1 illustrates how this will work when the user is playing Othello.

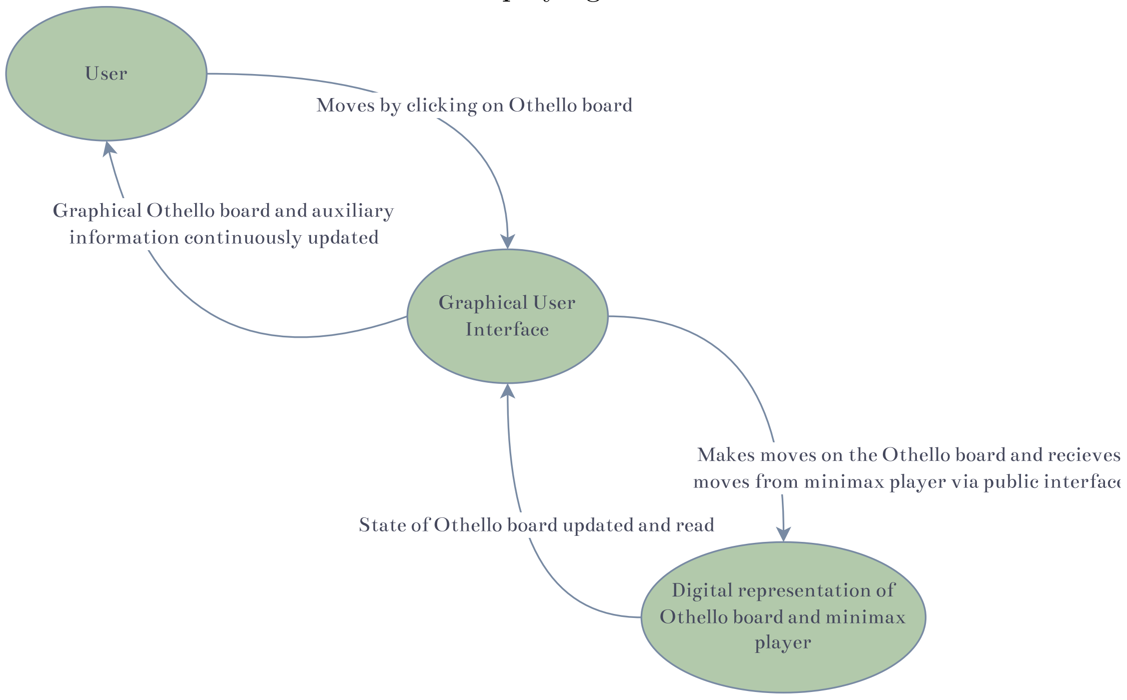

User
Moves by clicking on Othello board
Graphical Othello board and auxiliary information continuously updated
Graphical User Interface
Makes moves on the Othello board and recieves moves from minimax player via public interfac
State of Othello board updated and read
Digital representation of Othello board and minimax player

Figure 2.1: This is how data is transferred between the user, the graphical user interface, and the digital representation of Othello and the minimax player while the user is playing Othello.

2.1.1 Class Diagram

Figure 2.2 shows a Unified Modelling Language diagram of the classes and fields and methods which I will use in my project.

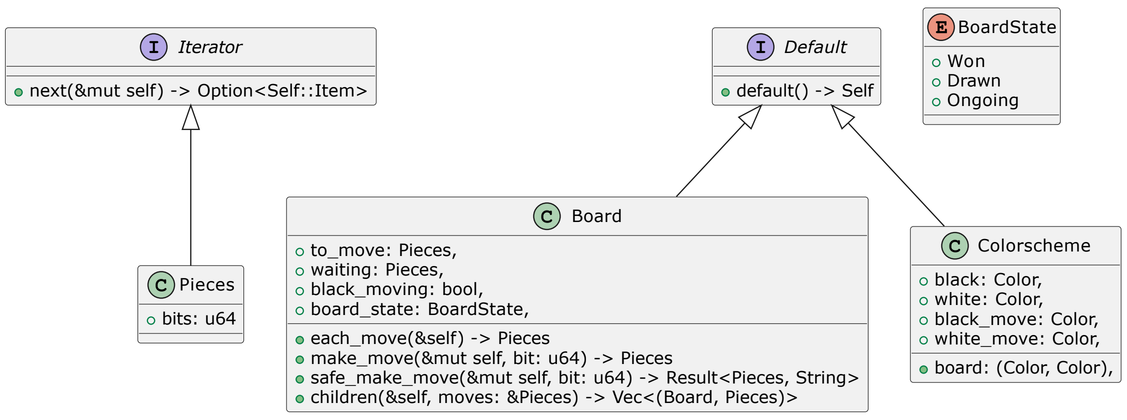


Figure 2.2: These are classes and fields and methods which I will need in order to complete the project

2.2 Othello

In this section, I will explain how I intend to implement a digital representation of Othello, including the data structures and algorithms which I am likely to need. I will not, however, discuss how I intend to implement the graphical user interface. I will include snippets of C-ish pseudocode to aid some of the explanations.

2.2.1 The Board

The pieces on the board can be thought of as an eight by eight 2D array with each row in the 2D array representing a row of eight squares on the board, and each element of that row representing either an empty square, a black disc, or a white disc. The starting position of Othello be declared like so:

```
enum Piece {
    Black,
    White,
    Empty
};

Piece board[8][8];
// Fill the board with empty squares first:
for (int i = 0; i < 8; ++i) {
    for (int j = 0; j < 8; ++j) {
        board[i][j] = Empty;
    }
}
// Set the four starting pieces:
board[3][4] = board[4][3] = Black;
board[3][3] = board[4][4] = White;
```

There are some problems with this method, e.g., consider how the empty squares would be determined:

```
bool empties[8][8];
for (int i = 0; i < 8; ++i) {
    for (int j = 0; j < 8; ++j) {
        empties[i][j] = (board[i][j] == Empty);
    }
}
```

Every square in the board has to be checked individually.

Bitboards

Luckily, there is a much more efficient way of encoding the board, called a bitboard. With a bitboard, the board can be encoded using two unsigned 64-bit integers. One of the integers stores the locations of the black discs, and the other, the white discs. An Othello board in the starting position could be declared like this:

```
// A bitboard is an unsigned 64-bit integer
typedef uint64_t bitboard;

bitboard blacks = 0, whites = 0;
blacks |= (1 << (3 * 8 + 4)) | (1 << (4 * 8 + 3));
whites |= (1 << (3 * 8 + 3)) | (1 << (4 * 8 + 4));
```

The `|` (pipe) operator means bitwise OR, and the `<<` operator means left shift. It is often helpful to think of the bitboards as sets of locations (e.g. $b = \{(3, 4), (4, 3)\}$), and the bitwise OR operator as the union ($\cup$) of two sets, and the bitwise AND operator (`&`) as the intersection ($\cap$) of two sets. To obtain a bitboard containing the locations of the empty squares, one can take the bitwise and of the complement of each board:

```
bitboard empties = (~blacks) & (~whites);
```

Almost all other operations on the board have a much faster analogue with bitboards too. Even if there is not a faster one, the bitboard can be treated as a 2D array by accessing individual bits via bitshifts as demonstrated below, so there is certainly no speed lost in using bitboards.

```
array_board[i][j] == (bit_board >> (8 * i + j))
```

Bitboards also use less memory than a 2D array, as each element in the 2D array occupies at least one byte in memory, so the 2D array uses at least 64 bytes in total. The two bitboards use $\frac{2\times64}{8} = 16$ bytes.

2.2.2 Iterating Over the Discs on the Board

This would be fairly trivial given a 2D array of pieces. It could be done like so:

```
int whites = 0, blacks = 0;
for (int i = 0; i < 8; ++i) {
    for (int j = 0; j < 8; ++j) {
        if (a[i][j] != 0) {
            do_stuff();
        }
    }
}
```

Note that each empty square must be checked, despite rarely being of interest, e.g. if one wanted to count the number of white discs on the board, it would be better to skip any empty squares.

I am using a bitboard, so fortunately there is a faster method. The least significant bit of an integer can be computed like so:

```
int num;
int least_significant_bit = num ^ (num & (num - 1));
```

The circumflex (^) operator means bitwise XOR. When a binary number is decremented, the trailing zeroes all flip to ones and the least significant one flips to a zero, so the binary representation of `num & (num - 1)` is the same as the binary representation of `num` except that the least significant bit of `num` is a zero in `num & (num - 1)`. This is the only difference. `num ^(num & (num - 1))` will be all zeroes except for at the least significant bit of `num`, because this is the only location where `num` and `num & (num - 1)` differ. The bits on the bitboard can be iterated over like so:

```
bitboard black;
int black_discs = 0;
while (black != 0) {
    ++black_discs;
    black &= black - 1;
}
```

The above snippet counts the number of bits on the bitboard `black` and stores the result in `black_discs`. Note that this time, the next disc is found directly instead of by iterating over empty squares until the next disc is found.

2.2.3 Computing the Legal Moves for an Othello Position

Finding all the legal moves from an Othello position efficiently is important because these must be used by minimax to find future positions. The number of legal moves is also a good indicator of how advantageous a position is for one player.

First, I must introduce the concept of sliding a bitboard. Sliding the bits on a bitboard in a particular direction (north, northeast, east, etc.) can be done by bitshifting the whole bitboard by a certain number of bits depending on the direction.

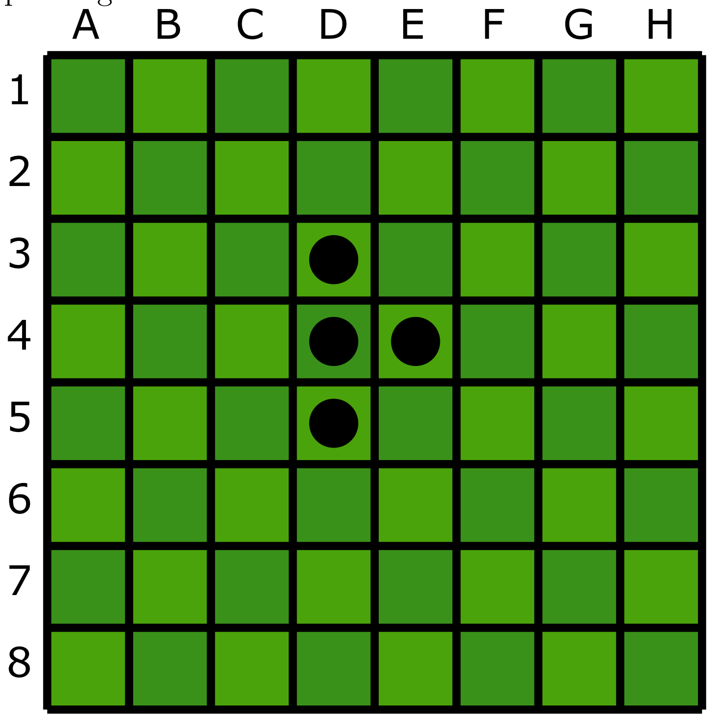


Figure 2.3: The black discs on the board before having been slid.

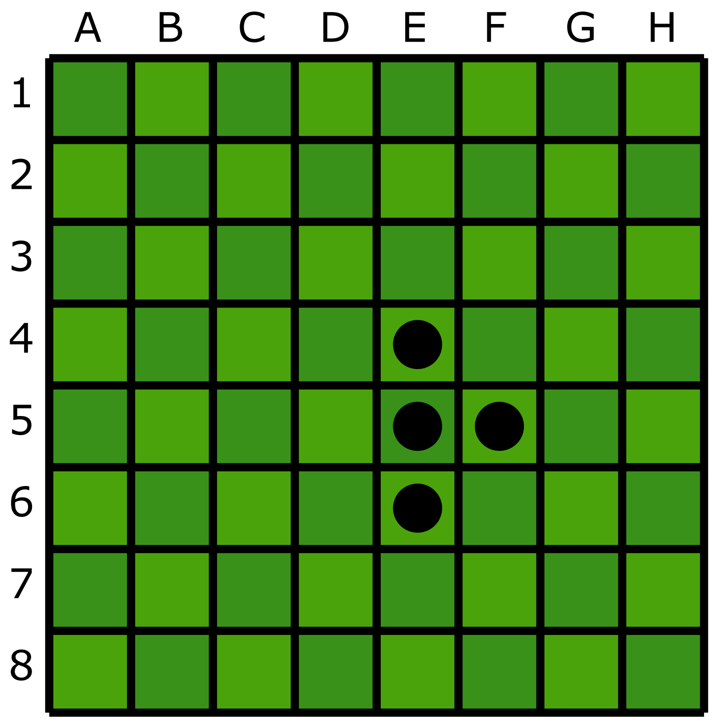


Figure 2.4: The black discs on the board after having been slid southeast.

In figure 2.4, the bits from the bitboard which represents the board shown in figure 2.3 have been slid to the southeast. If these are the positions of each bit:

```
63 62 61 60 59 58 57 56
55 54 53 52 51 50 49 48
47 46 45 44 43 42 41 40
39 38 37 36 35 34 33 32
31 30 29 28 27 26 25 24
23 22 21 20 19 18 17 16
15 14 13 12 11 10 09 08
07 06 05 04 03 02 01 00
```

with the 0^{th} bit being the least significant (in the ones column), then the bit to the southeast of the n^{th} bit is the $(n-9)^{\text{th}}$ bit. In other words, to translate the bits on the bitboard one step to the southeast, the bitboard must be right shifted by nine bits.

Masking

Bits on the lower or right-hand-side edge of the board have no bit to their southeast. Bits on the bottom edge (positions 00 - 07) will disappear once they are right shifted by nine, but bits on the right-hand-side edge of the board will not. In order to solve this, I will take the bitwise AND of the board with a bitboard representing the valid positions from which to shift bits before I shift the bitboard. I will store the complement of this bitboard, i.e., the bits which should *not* be shifted. This is called a mask.

Direction	Left shift by
North	+8
Northeast	+9
East	+1
Southeast	-7
South	-8
Southwest	-9
West	-1
Northwest	+7

Now that the pieces on the board can be slid, I will explain how the bitboard of legal moves can be computed efficiently given bitboards describing a position.

In Othello, a legal move is one which flips at least one of the opponent's discs. This is done by capping an unbroken line of the opponent's discs with a piece at each end. Each of the opponent's discs in that aforementioned line is flipped, changing color. The legal moves of a position can be generated by iterating through the directions (north, northeast, etc.) and sliding the discs of the player who is about to move by one square in that direction, then taking the bitwise AND (analogous to a set intersection) with the opponent's discs, then sliding *those* bits once more in that direction and taking the bitwise AND with the empty squares (after all, a move must be made on an empty square). The bits left represent the legal moves. Repeating this for each direction until there are no bits left generates all the legal moves. This is clearly much faster than iterating over all 64 squares on the board and checking whether each of them is legal.

```
bitboard moves = 0;
for each (direction, mask) {
    candidates = opponent & ((player & ~mask) << direction);
    while (candidates != 0) {
        moves |= empty & ((candidates & ~mask) << direction);
        candidates = opponent & ((candidates & ~mask) << direction);
    }
}
```

`mask` is a variable solving the problem described in 2.2.3.

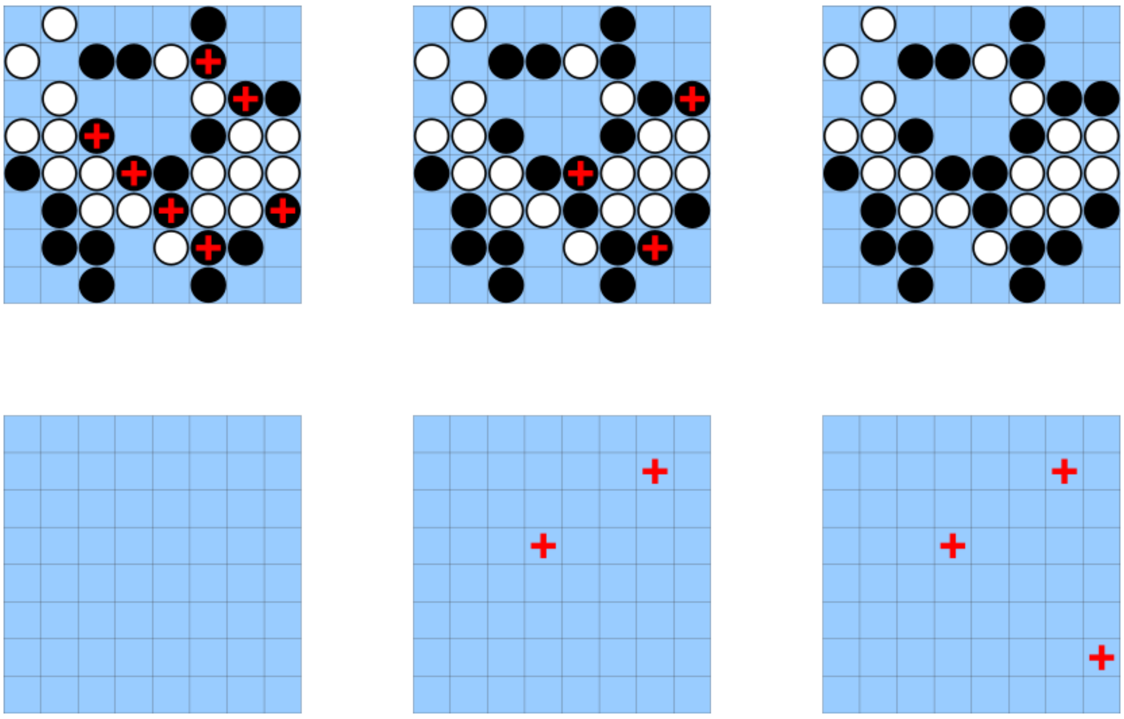


Figure 2.5: Legal moves for white which have been generated by shifting east. The top row shows the discs and the red crosses are the candidates. The bottom row shows the legal moves which have been accumulated.

In figure 2.5, east is the direction being considered. The red crosses in the top row are the `candidates` from the pseudocode above, and the red crosses in the bottom row are the `moves`. When there are no more candidates, i.e., there are no more red crosses in the top row, east stops being considered and the next direction is considered.

2.2.4 Computing the Frontier Squares for an Othello Position

A frontier square for some player is an empty square adjacent to at least one of that player's discs. Having many frontier squares is generally undesirable because all of the opponent's legal moves are on frontier squares, and legal moves must flip some discs to the opponent's color. Having no frontier squares would mean the opponent had no legal moves, and they would have to miss their turn. For this reason, the number of frontier squares is a useful metric for the evaluation function to use. Positions with many frontier discs should be avoided.

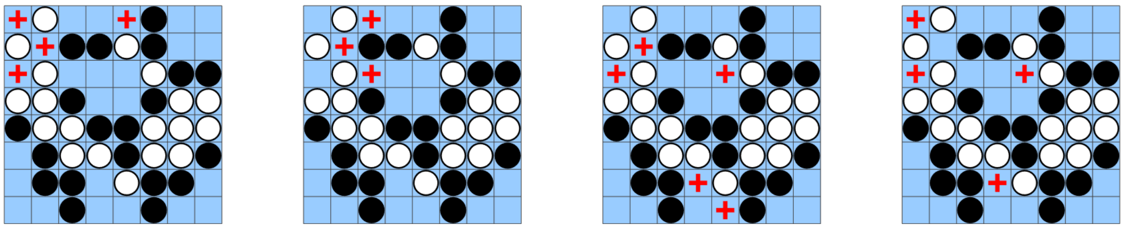


Figure 2.6: The frontier discs for white can be computed by sliding whites pieces in each direction by one square (only north, east, south and west are included here, going from left to right) and intersecting (bitwise AND) with the empty squares. The red crosses are the frontier squares in one direction.

```
bitboard opponent;
bitboard player;
bitboard frontier = 0;
for each direction {
    bitboard empty = ~(player | opponent);
    frontier |= (player << direction) & empty;
}
```

The pseudocode above demonstrates how a bitboard containing the frontier discs for `player` could be computed.

2.2.5 Minimax

As mentioned in 1.3, minimax is a brute force algorithm which uses the evaluation function to evaluate every position which could possibly occur after exactly (not before) some number of moves (called the search depth), and then evaluates the positions which occur before the search depth using logic. This is then used to evaluate the current position assuming perfect play from both players. One player is called the minimizing player and the other is called the maximizing player (hence *minimax*). Lower evaluations are better for the minimizing player and greater evaluations are better for the maximizing player. Consider the game tree as in 1.3, but instead of the nodes being Othello positions, they are evaluations of Othello positions.

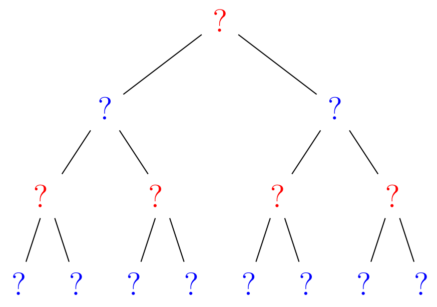


Figure 2.7: The initial tree before anything has been filled in by minimax.

Suppose red is the minimizing player and blue is the maximizing player. The color of a node is the player whose move it is at that node, so red moves first. To decide whether to choose the left or the right branch, red must know what the best (greatest for blue) evaluation which blue can force is for each branch, and choose the least because red is the minimizing player. To compute the greatest evaluation which blue can force from the left node, the least evaluations which red can force from its two children must be computed, and so on. This is why minimax is a recursive algorithm; to evaluate a position using minimax, its children must be evaluated with minimax first unless the search depth is zero. When the search depth is zero, i.e., the node being considered is a leaf node, the position is evaluated using the evaluation function discussed in 3.2.8. Once the evaluations for each node in the left subtree has been evaluated, the tree looks like this:

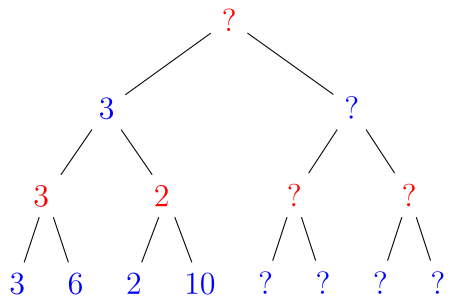


Figure 2.8: Now the left subtree has been filled in.

The leftmost red node is 3 because red would choose 3 from that position as $3 < 6$. The same reasoning goes for the other red node, and the blue node which is the parent of those two red nodes is evaluated at 3 because blue is the maximizing player so would choose the 3 instead of the 2. This is the result after filling in the rest of the tree:

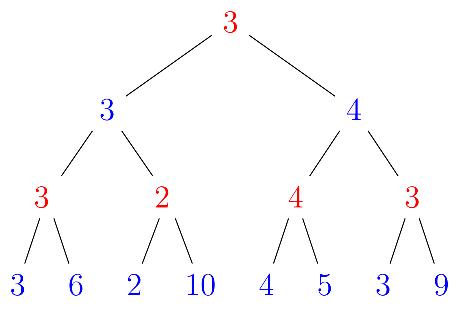


Figure 2.9: Now the evaluation of every node in the tree has been calculated.

The root node has been evaluated at 3, meaning that no matter what blue does, red can always force a position which has an evaluation of at most 3. It can be said for certain that the move which yields the least (best) evaluation for red in three moves is moving to the left branch of the tree. This is how minimax can be used to compute good moves in a two-player game such as Othello.

```
fn minimax(static_eval, position, maximizing, depth) {
    child_positions = children(position);
    if empty(child_positions) or depth == 0 {
        return static_eval(position);
    }
    if maximizing {
        best_eval = -∞;
        for each child in child_positions {
            best_eval = max(best_eval,
                            minimax(static_eval,
                                    child,
                                    is_maximizing(child),
                                    depth - 1));
        }
    }
    else if not maximizing {
        best_eval = ∞;
        for each child in child_positions {
            best_eval = min(best_eval,
                            minimax(static_eval,
                                    child,
                                    is_maximizing(child),
                                    depth - 1));
        }
    }
    return best_eval;
}
```

2.2.6 Minimax with alpha-beta pruning

Alpha-beta pruning is an optimization for minimax which allows branches of the game tree to be "pruned", meaning time can be saved not searching them because it is impossible that any of them will be relevant given perfect play. Minimax assumes perfect play because if the opponent plays perfectly (perfectly according to our evaluation function that is), that has been assumed, and if the opponent does not play perfectly, this does not need to be planned for because it is advantageous anyway.

Optimizing minimax with alpha-beta pruning

In the example above, some positions were evaluated unnecessarily (assuming the bottom row of nodes were evaluated from left to right).

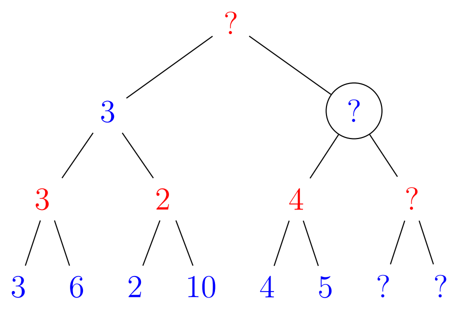


Figure 2.10: Enough nodes have been evaluated for the best move to be determined.

In the tree above, one of the two children of the circled node has been evaluated. Blue is the maximizing player so will pick the greatest number, which will be at least 4. With this knowledge, evaluating the rest of the nodes is unnecessary because the last blue question mark must be at least 4, but red is the minimizing player and so is guaranteed to go down the left branch as red can guarantee an evaluation of 3 that way. This is how alpha-beta "prunes" subtrees from having to be searched at all.

To do this, two variables, called α and β, are the best evaluations which black and white can force, respectively. If $\alpha \geq \beta$, the evaluation of the current position stops, and the best evaluation which that player can achieve so far is returned. Consider the example above. When the circled position is being evaluated, $\alpha = 4$ and $\beta = 3$ because from the circled node, blue can choose the 4 and red can choose the 3 from the root node.

```
fn minimax(static_eval, position, maximizing, depth, alpha, beta) {
    child_positions = children(position);
    if empty(child_positions) or depth == 0 {
        return static_eval(position);
    }
    if maximizing {
        best_eval = alpha;
        for each child in child_positions {
            best_eval = max(best_eval,
                            minimax(static_eval,
                                    child,
                                    is_maximizing(child),
                                    depth - 1,
                                    alpha,
                                    beta));
            alpha = best_eval;
            if beta <= alpha {
                break;
            }
        }
    }
    else if not maximizing {
        best_eval = beta;
        for each child in child_positions {
            best_eval = min(best_eval,
                            minimax(static_eval,
                                    child,
                                    is_maximizing(child),
                                    depth - 1,
                                    alpha,
                                    beta));
            beta = best_eval;
            if beta <= alpha {
                break;
            }
        }
    }
    return best_eval;
}
```

2.2.7 Making a Move

In Othello, play alternates between two players unless the player whose turn it is does not have a legal move, then their opponent gets to make another move if they have any legal moves, and if neither player has a legal move, the game ends and the winner is determined by counting the number of discs which each player has. After each move some of the opponent's discs will be flipped (otherwise the move would not be legal). I need to implement a function to let me easily make moves on the bitboard. First, I will explain how I flip the discs after a move.

I am going to use a bitboard, so I will use the sliding mechanism explained in 2.2.3 to make this function as efficient as possible. When a player places a disc, I slide that disc (represented as a single bit) in each direction one square at a time while it is on top of one of the opponent's discs, accumulating the discs it passes over in a separate bitboard. When it is not on an opponent's disc, I check whether it is on the player's disc. If so, the discs which are stored in that separate bitboard get flipped, otherwise they do not. Here is some pseudocode for this algorithm.

```
// move is the variable encoding the move as a single bit on a bitboard.
// board is the board on which this move will be made.
for each (direction, mask) {
    // line is the separate bitboard onto which discs which might be flipped are
    accumulated.
    bitboard line = 0; // as a bitboard is an integer, 0 means empty
    bitboard sliding_bit = (move & ~guard) << direction;
    // This condition checks whether the moving bit is still on top of the opponent
    's discs.
    while moving_bit & board.opponent != 0 {
        line |= sliding_bit; // Insert the current square onto the end of the line
        sliding_bit = (sliding_bit & ~guard) << direction;
    }
    // This condition checks if the sliding bit is on top of one of the player's (
    not the opponent's) discs.
    if sliding_bit & board.player != 0 {
        board.player |= line;
        board.opponent &= ~line;
    }
}
```

Having flipped the correct discs, I now need to determine whose move it is. I first assume that it is the opponent's turn. If the opponent has any legal moves, then it is the opponent's turn and I am done. If the opponent has no legal moves, I assume it must be the player's turn. If the player has any legal moves it is their turn. If not, the game is over, and the winner is determined.

```
// discs already flipped; now whose move it is is determined
swap(board.player, board.opponent)
if legal_moves(board) != 0 {
    // I am assuming this will be inside a function for making a move, so I can
    return
    return;
}
swap(board.player, board.opponent)
if legal_moves(board) != 0 {
    return;
}
```

```
// Game must be over
if count(board.player) < count(board.opponent) {
    // board.won == true means that board.player won, board.opponent lost.
    board.won = true;
}
if count(board.opponent) < count(board.player) {
    swap(board.player, board.opponent)
    board.won = true;
}
board.drawn = true;
```

2.2.8 Static Evaluation Function

The minimax algorithm is useless without a static evaluation function. A static evaluation function takes an Othello position and returns a higher value if that position is good for one player, and a lower value if that position is better for the other player. I will make it return a higher value for good positions for black and a lower value for good positions for white, because black moves first and it is conventional to return higher values for good positions for the player who moves first. The evaluation method I chose in 1.3.2 combines mobility based evaluation and disc square evaluation. Mobility based evaluation takes the number of frontier squares and legal moves into account. It does not really make sense to talk about the number of legal moves for the player whose move it is *not* in a position. This means that the evaluation function will not necessarily return zero for a position which is just as good for black as it is for white, because it only takes into account the number of legal moves for the player whose turn it is but not the number of legal moves for the player whose turn it is not, so it is asymmetric in some sense. I know how to compute the number of legal moves and the number of frontier discs, so the mobility based part of my evaluation function is taken care of. I can take the bitwise AND of a player's discs with a set of discs which are important to occupy (I will use the corners) for the disc square table part of the evaluation function. I can count the number of discs on a bitboard efficiently using the algorithm explained in 2.2.2, and I will use this to count the number of discs in the corners and the number of legal moves and the number of frontier discs, so I will be able to implement my static evaluation function.

2.3 User Interface Design

The design of the user interface is important for this project because most of the end user's objectives relate to the user interface, as he expects to play Othello with a graphical user interface and not with a command line interface.

2.3.1 Main Menu

Figure 2.11 shows how the main menu will allow the user to customize everything which they mentioned in the interview transcribed in 1.4.1. I plan to include a difficulty adjustment, a button to switch between playing black and white, a button to open another window in which the colors of the board can be customized, and a button allowing the user to start playing Othello, which will open another window showing an Othello board.

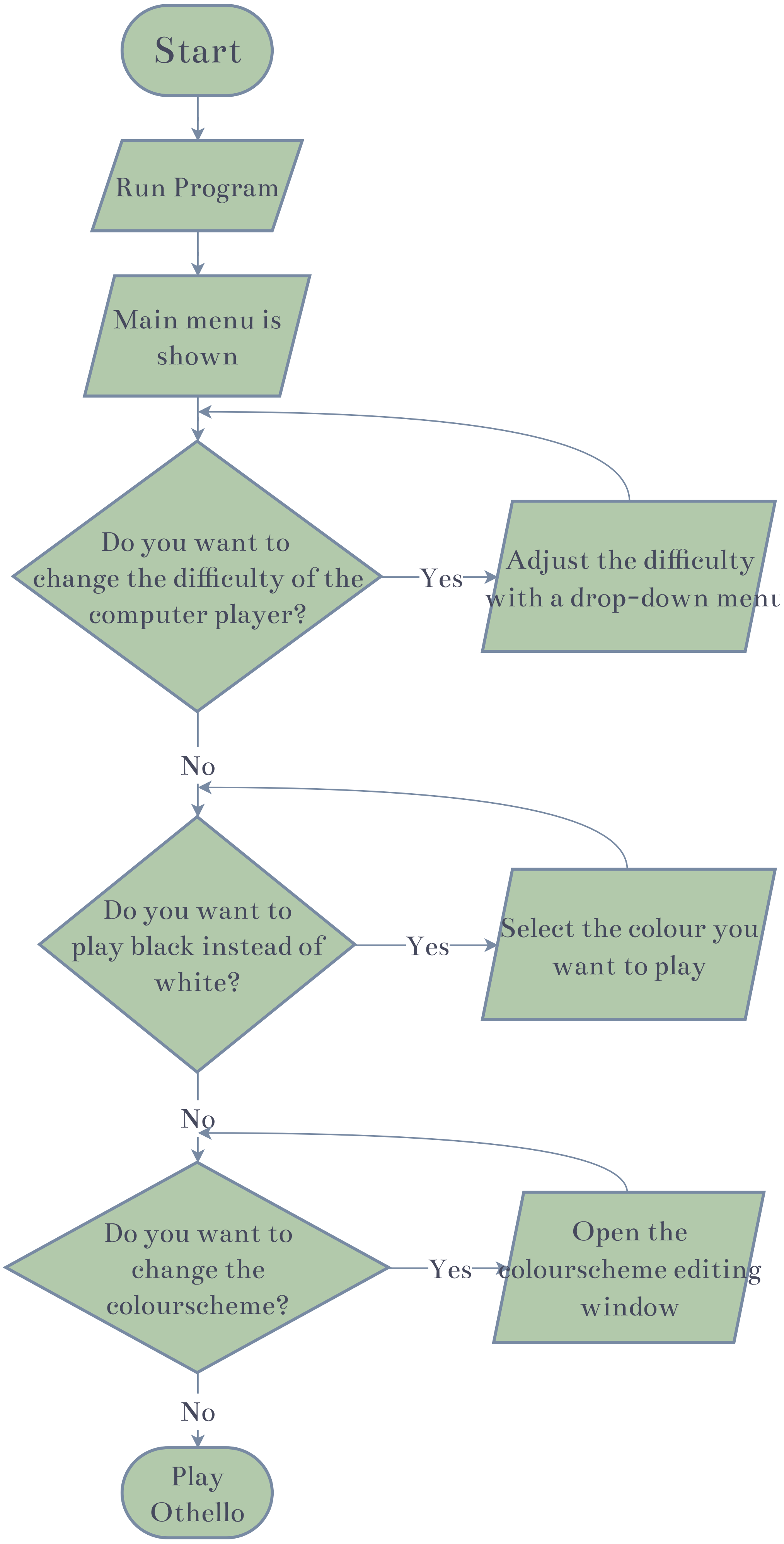

Start
Run Program
Main menu is shown
Do you want to change the difficulty of the computer player?
Yes
Adjust the difficulty with a drop-down menu
No
Do you want to play black instead of white?
Yes
Select the colour you want to play
No
Do you want to change the colourscheme?
Yes
Open the colourscheme editing window
No
Play Othello

Figure 2.11: This flowchart shows how the user would select different options before the Othello game with the main

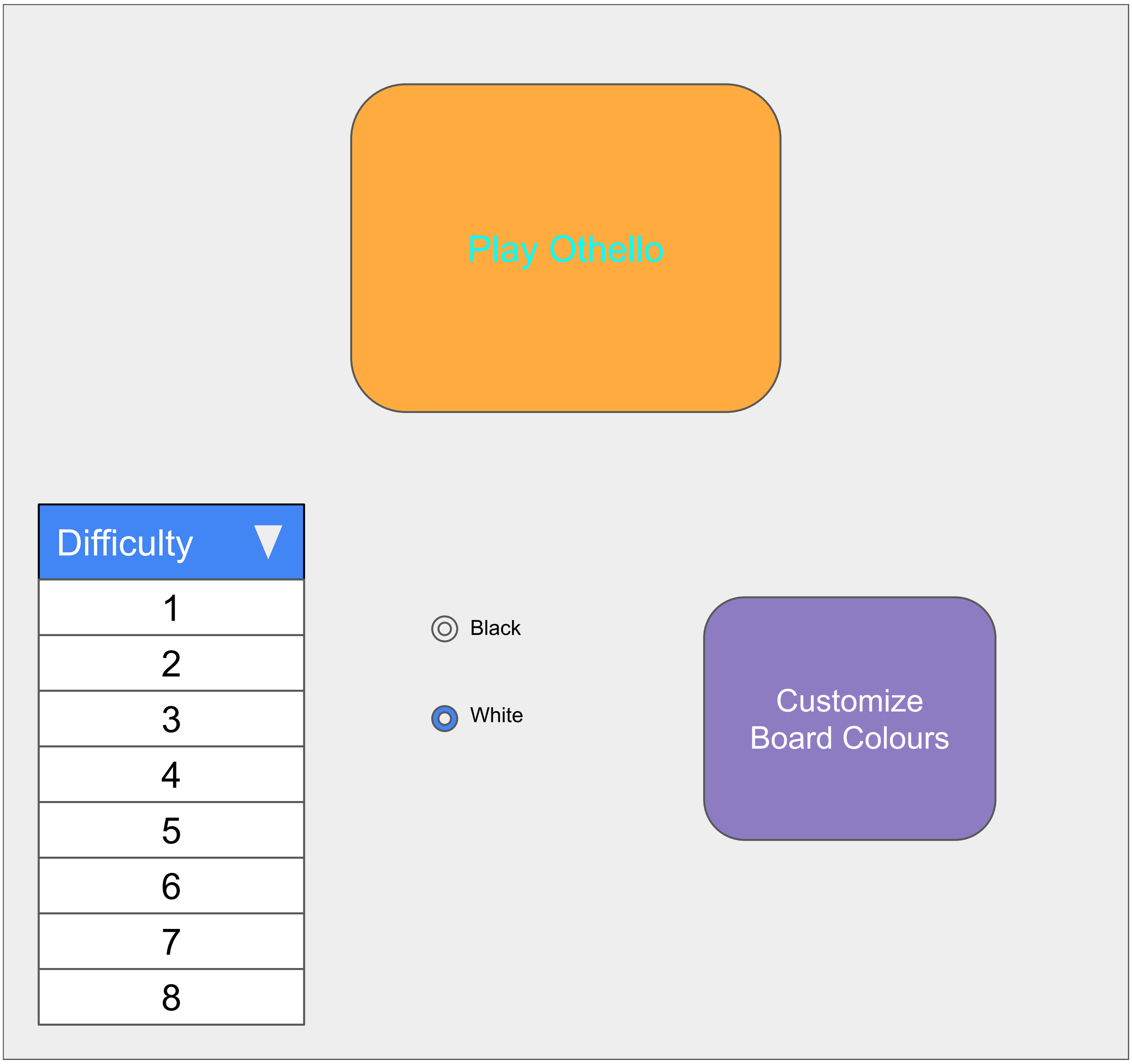


Figure 2.12: A demonstration of the buttons I want to include on the main menu. The player can see that they will play the white pieces in the current state.

Implementing this main menu correctly will fulfill objectives one, three, four, and five.

2.3.2 Board color Customization

This window fulfills objective five. I want the user to choose the colors of the board and discs with six color pickers, each controlling one of the six colors used to draw the board, which are the color of the "black" discs, the color of the "white" discs, the color of half of the squares on the board, the color of the other half of all squares on the board, the color in which a legal move for black is shown, and the color in which a legal move for white is shown.

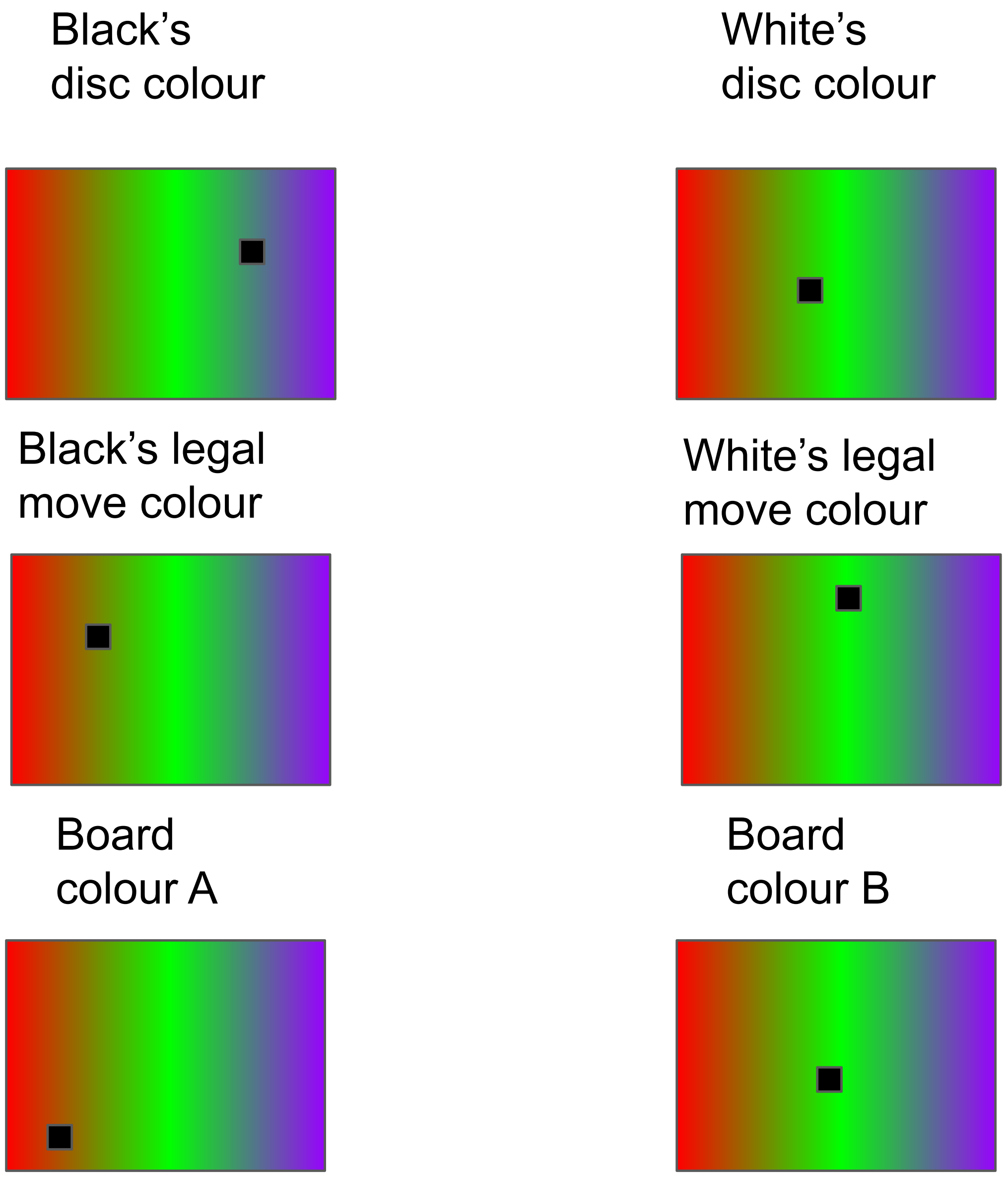


Figure 2.13: This is the window which the player will use to choose the color of the board.

2.3.3 Othello Board Graphical User Interface

Figure 2.14 shows how the graphical user interface will appear to the player when they are playing Othello.

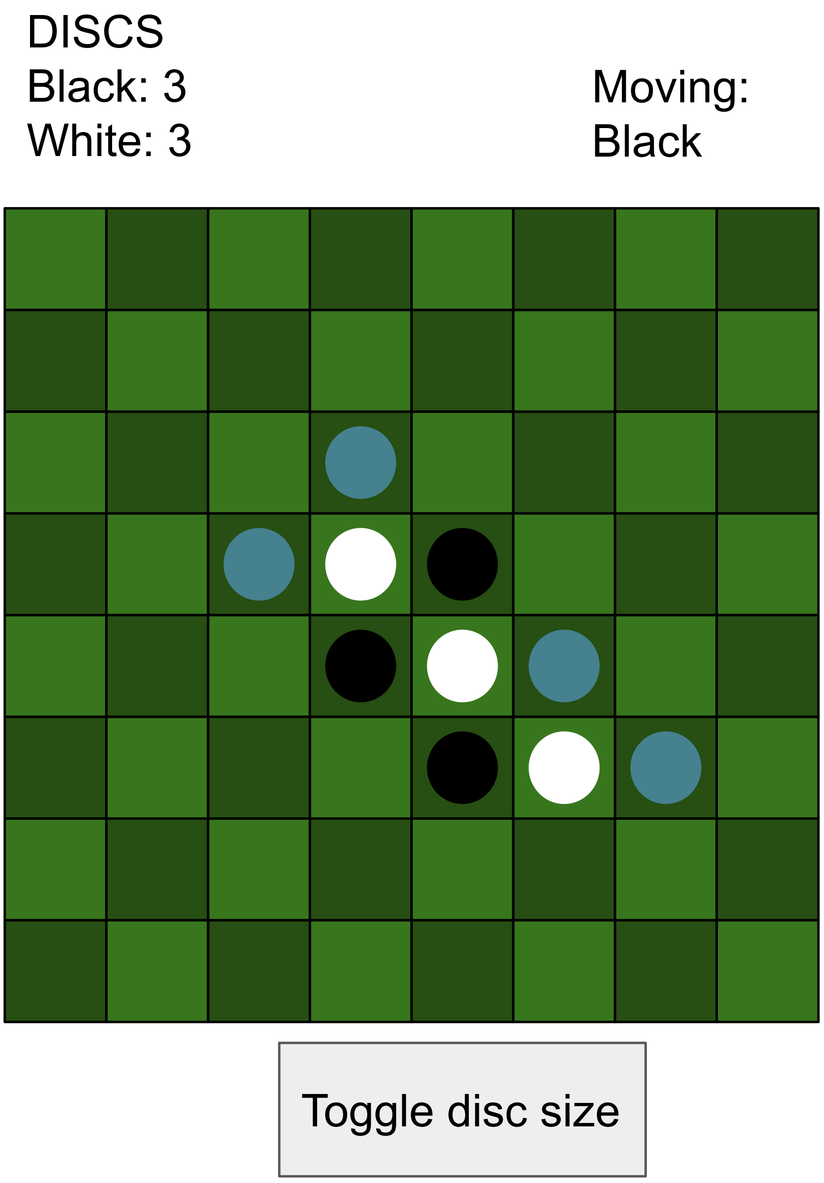


Figure 2.14: This is something like what the player will see when they are playing Othello

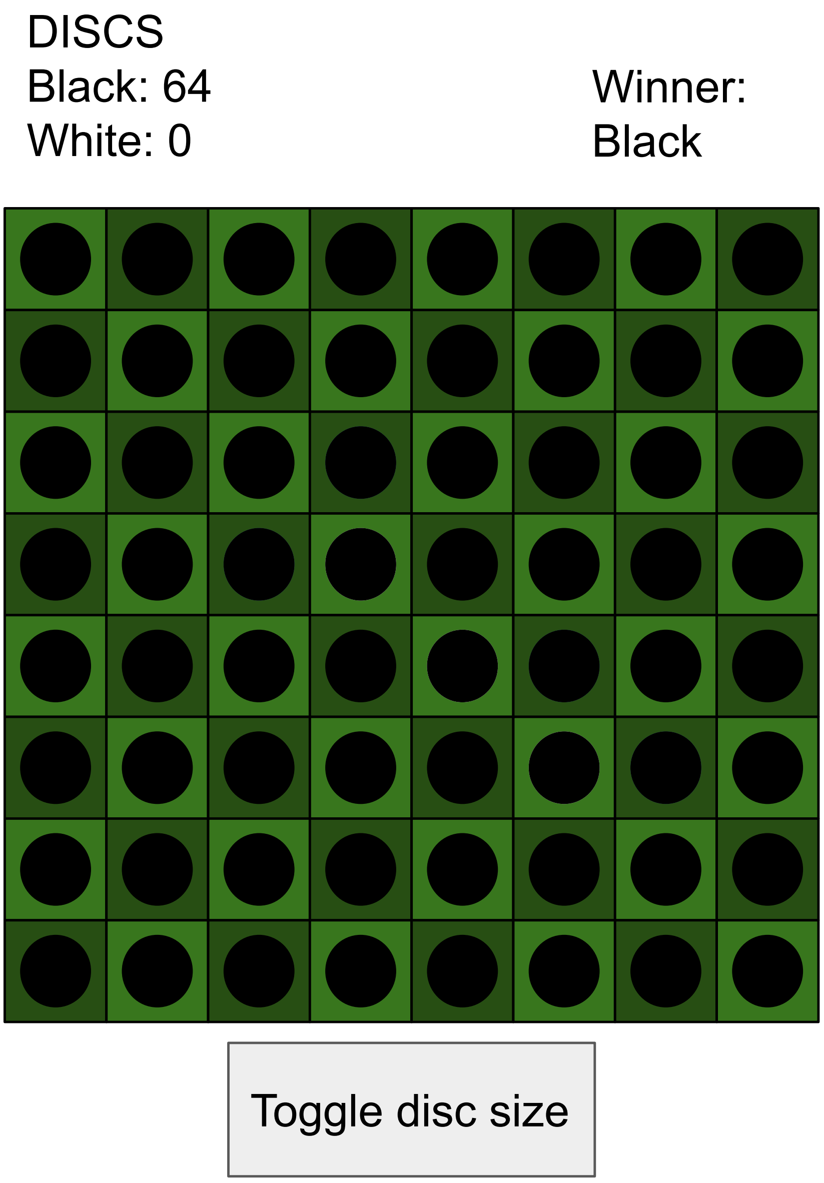


Figure 2.15: Now the labels at the top have changed to show that black has won.

Implementing this Othello window correctly will fulfill objectives two, six, seven, eight, nine, and ten.

2.3.4 Multi-threading

Objective nine says: *The graphical user interface must remain responsive while the computer player is computing a move.* I need to use multi-threading in order to fulfill this objective. In a single-threaded program, each step in the program is done serially, i.e., one after the other. The computer player might take more than half a second to compute a move depending on the minimax depth, so if each step one done after the other, when the user clicks on a button while the computer player is computing a move, the software could take too long to respond because the user would have to wait for the computer player to make the move before their click appears to be registered. In a multi-threaded program, some steps can be done concurrently. This allows the computer to spend some resources on checking for clicks, and some different resources on the computer player, so that the user can interact with the graphical user interface while the computer is making a move.

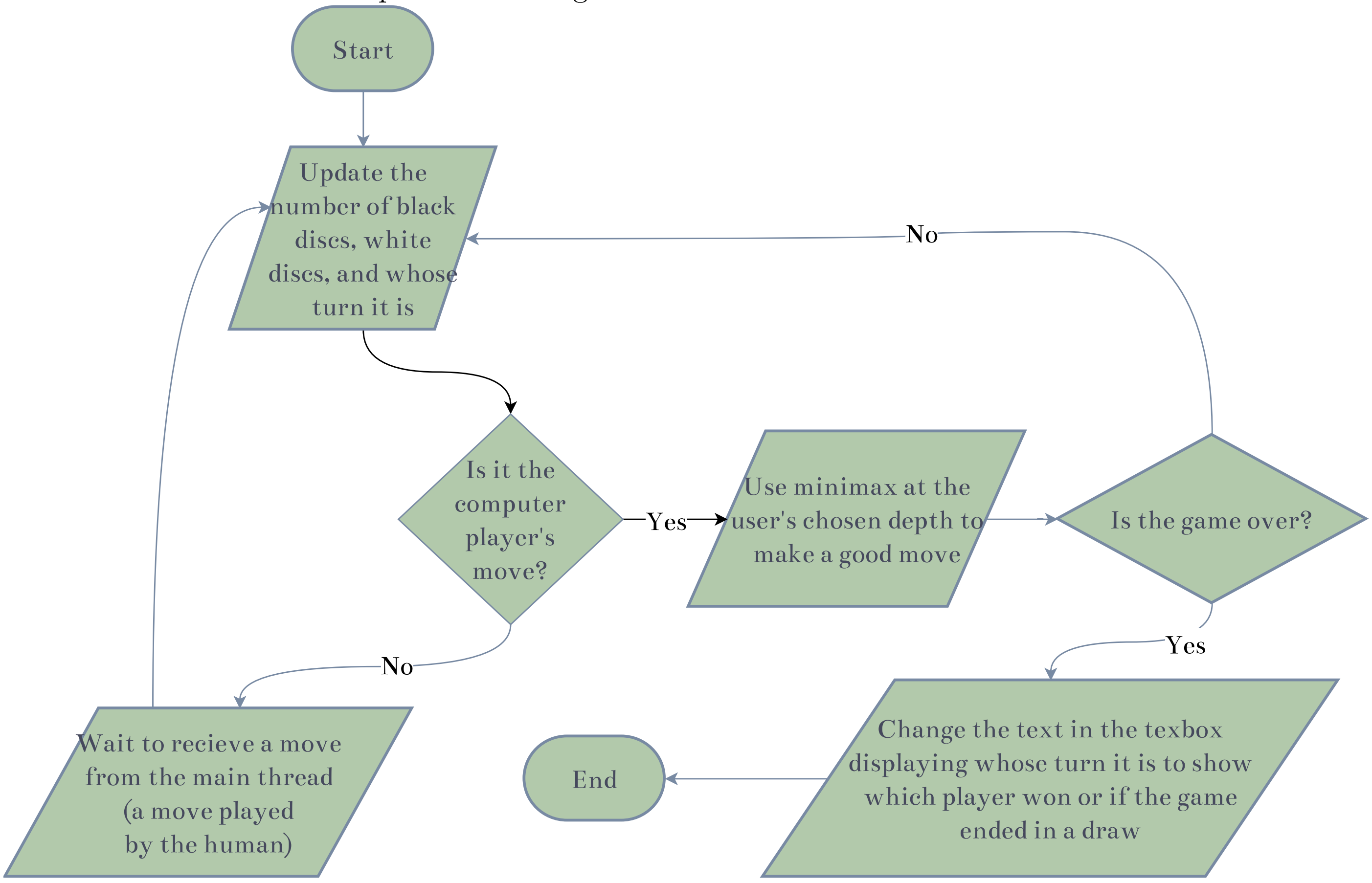

Start
Update the number of black discs, white discs, and whose turn it is
Is it the computer player's move?
Yes
Use minimax at the user's chosen depth to make a good move
Is the game over?
No
Yes
No
Wait to recieve a move from the main thread (a move played by the human)
Change the text in the texbox displaying whose turn it is to show which player won or if the game ended in a draw
End

Figure 2.16: This is what my second thread will do when the user is playing Othello.

The main thread will check for user input via the graphical user interface and redraw the board each time it changes. This will mainly be handled by FLTK (the GUI library which I plan to use). The second thread will update the counts of black and white discs and whose move it is, shown in the graphical user interface, every time a move is made by either the computer player or the human player. The second thread will also call minimax when it the computer player's turn. I have assigned each thread these roles in particular because neither thread will need to do two things at once. When the second thread is computing a move, no moves are being made, so the display of the counts of black and white discs and whose move it is do not need updating.

Mutexes and Channels

Data needs to be sent between the two threads, e.g. when the human makes a move, which is received by the main thread, the second thread needs to know what that move was in order to maintain the correct position on the Othello board. I plan to do this in two ways.

A mutex guards some data which can be read and written to by multiple threads. Only one thread can read and write at any one time, and the thread which can read and write to the mutex at some time is said to have a lock on the mutex. For another thread to read or write to the mutex, they have to acquire a lock on the mutex, and this can only happen when the thread which currently holds a lock for the mutex drops the lock, i.e., the lock goes out of scope. A mutex could be used to allow both the main thread and the second thread to read and write to the board.

A channel consists of two objects: a sender, and a receiver. A thread can send data, using the sender object, to another thread which holds the receiver object. All the data sent via the channel is stored in chronological order until the data is received. This could be used in order for the main thread to send moves made by the human to the second thread.

2.3.5 Drawing the Othello Board with FLTK

FLTK has functions for drawing straight lines and rectangles and circles. I will use these to draw the Othello board in the graphical user interface. An Othello board consists of sixty-four squares, possibly with discs on them. I will draw the boundaries of the squares with horizontal and vertical straight lines. I will draw the squares themselves by drawing filled rectangles which have the same width and height. I will draw the discs as filled circles. I also want to chequer the colors of the squares on the board to make the boundaries of the squares more obvious.

If each square on the board is l pixels on each side, the square in the (zero-indexed) i^{th} column and the j^{th} row with its top left pixel $i \times l$ pixels to the right and $j \times l$ pixels down from the top left corner of the board.

```
boardcolor1 = color::darkgreen;
boardcolor2 = color::lightgreen;
square_size = 50; // 50 pixels is the side length of one of the square on the
    Othello board
board = get_position(); // board stores the current position on the Othello board
for i = 0 to i = 7 {
     for j = 0 to j = 7 {
          boardcolor = boardcolor2;
          if i + j % 2 == 0 {
               boardcolor = boardcolor1;
          }
          // Supposing the first two arguments are the x and y coordinates of the top
     left corner of the rectangle, and the second two are the dimensions of the
    rectangle,
          // and the last is the color with which to fill the rectangle.
          draw_rectangle(i * 50, j * 50, 50, 50, boardcolor);
          if board.get_disc(i, j).exists() {
               disccolor = color::white;
               if board.get_disc(i, j).is_black() {
                    disccolor = color::black;
               }
               // Supposing the first two arguments are the coordinates of the centre
    of the circle,
               // the third argument is the radius of the circle, and the fourth
    argument is the color with which to fill the circle
               draw_circle(i * square_size, j * square_size, sqare_size / 3, disccolor
    );
          }
     }
}
board_size = 8 * square_size;
for i = 0 to i = 8 {
     // Supposing the first four arguments are the two coordinates between which the
     line segment will be drawn
     draw_line(0, i * square_size, board_size, i * square_size, color::black); //
    Draw a horizonatl line
     draw_line(i * square_size, 0, i * square_size, board_size, color::black); //
    Draw a vertical line
}
```

The pseudocode above demonstrates how I will use the functions available from FLTK to draw the Othello board.

2.4 Techniques Used

Technique	Description	Location
Bitboards	Representing the pieces on the board with two binary integers, using bitwise operations to access and modify pieces on the board.	The struct which uses a bitboard is `Pieces`, implemented in 3.1.2.
Alpha-beta pruning	Search every position in the game tree within a fixed number of moves from the current position, and prune subtrees which are guaranteed not to matter.	Minimax is explained in 2.2.5 implemented in 3.2.9.
Multi-threading	Run the computer player and GUI simultaneously in separate threads.	Multi-threading is explained in 2.3.4 and implemented in 3.4.2.

Chapter 3

Technical Solution

I will explain the data structures I used first, then the algorithms. This is because an understanding of the data structures used is needed to understand some of the algorithms.

3.1 Data Structures for Othello

Here I will explain all of the types I defined for the digital representation of Othello, but *not* for the GUI.

3.1.1 `BoardState`

A variable of type `BoardState` is either `BoardState::Won`, `BoardState::Drawn`, or `BoardState::Ongoing`. `BoardState` is implemented as an `enum` like so:

```
#[derive(Clone, PartialEq, Debug, Copy)]
pub enum BoardState {
    Won, // to_move has won, so we don't need a `BoardState::Lost`
    Drawn,
    Ongoing,
}
```

the first line tells Rust to automatically implement some useful methods.

3.1.2 `Pieces`

The `Pieces` struct is a wrapper for an unsigned 64-bit integer, on top of which I have implemented the `Iterator` trait which allows me to iterate over each bit in the integer.

```
#[derive(Clone, Copy)]
pub struct Pieces {
    pub bits: u64,
}

impl Iterator for Pieces {
    type Item = u64;
    fn next(&mut self) -> Option<Self::Item> {
        if self.bits == 0 {
            None
        } else {
            let bit = self.bits ^ (self.bits & (self.bits - 1));
            self.bits &= self.bits - 1;
            Some(bit)
        }
    }
}
```

`Iterator` is a trait (similar to an interface) defined in Rust's standard library which can be implemented by defining the `next` method. Implementing `Iterator` for a type allows objects of that type to be used in for loops, and gives the programmer access to many other methods, e.g. `filter` and `map`. The `next` method returns the least significant bit and mutates the inner integer. This is all that is required to be able to iterate over each piece in an object which is of type `Pieces`, and all the methods on a type which implements the `Iterator` trait are available. When the object has no pieces left, i.e., `self.bits == 0`, a null value is returned, which is expressed in the return type of the method which is `Option<Self::Item>` and not just `Self::Item`. In `Self::Item`, `Self` is a synonym for `Pieces`, and `Item` is defined as an unsigned 64-bit integer on the seventh line.

3.1.3 `Board`

The `Board` struct combines the `Pieces` and `BoardState` types to store the state of an Othello board, using bitboards. There is also a flag, `black_moving` which indicates which color is moving.

```
#[derive(Clone, Copy)]
pub struct Board {
    pub to_move: Pieces,
    pub waiting: Pieces,
    pub black_moving: bool,
    pub board_state: BoardState,
}
```

3.2 Algorithms for Othello

Here I will explain all of the functions I defined when writing the digital representation of Othello.

3.2.1 Implementation of `std::default::Default` for `Board`

`std::default::Default` is a trait in Rust's standard library which requires one method, `default`, and only exposes one method, `default`. The `default` method is a constructor for the type which implements it, conventionally returning a "default" object of that type. In my case, I have implemented `default` to return an Othello board in a game's initial position, like the one shown in 1.1.

```
impl Default for Board {
    fn default() -> Self {
        Board {
            black_moving: true,
            board_state: BoardState::Ongoing,
            waiting: Pieces {
                bits: 0b1000000001000000000000000000000000000,
            },
            to_move: Pieces {
                bits: 0b100000010000000000000000000000000000,
            },
        }
    }
}
```

The bits field is filled in for each color in the starting position of an Othello board, which I worked out using an online bitboard viewer: `https://tearth.dev/bitboard-viewer/`. In Othello black moves first, so the `black_moving` field is set to `true`.

3.2.2 Implementation of `std::fmt::Debug` for `Board`

`std::fmt::Debug` is a trait, which when implemented for a type allows objects of that type to be printed to the terminal for debugging purposes. I could have used the defualt implementation with `#[derive(Debug)]` above the declaration of `Board`, but this would print the fields of the board one-by-one, so I would not be able to easily tell the state of my board and see if anything unexpected was happening before I implemented my GUI. By implementing the `Debug` trait myself, I can get a graphical visualization of the current state of the board in the terminal by printing colored block characters. A screenshot of what this function prints when called is shown in figure 3.1.

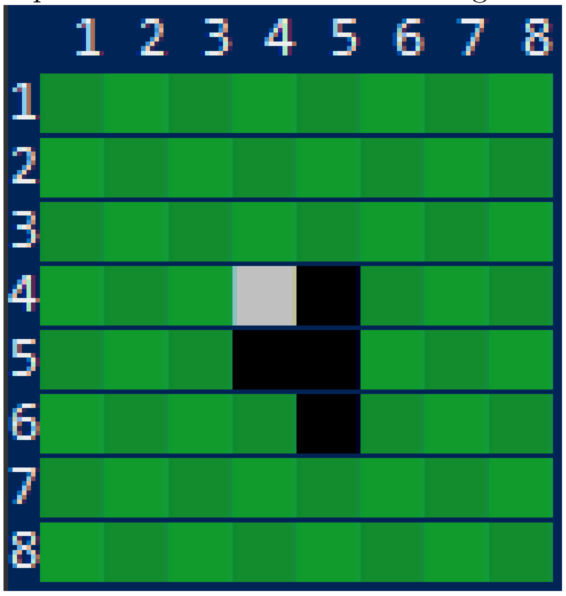


Figure 3.1: This is what my implementation of `Debug` prints when called.

3.2.3 Implementation of `Board::each_move`

`Board::each_move` is one of the methods of `Board`. It takes a read-only reference to the object of type `Board` which it is called on, and returns a `Pieces` object representing the locations of all the legal moves on the current state of the board.

The algorithm for computing the legal moves of an Othello position is explained in 2.2.3.

The Code

```rust
impl Board {
// The second item of each tuples are the squares where you shouldn't shl from for each dir
    pub const DIRECTIONS: [(i8, u64); 8] = [
        (8, 0),
        (9, 0x8080808080808080),
        (1, 0x8080808080808080),
        (-7, 0x8080808080808080),
        (-8, 0),
        (-9, 0x0101010101010101),
        (-1, 0x0101010101010101),
        (7, 0x0101010101010101),
    ];
    pub fn each_move(&self) -> Pieces {
        let (friend, opponent) = (&self.to_move.bits, &self.waiting.bits);
        let safe_shl = |a: u64, b: i8| if b > 0 { a << b } else { a >> (-b) };
        let mut moves: u64 = 0;
        let empty = !(friend | opponent);
        for (dir, guard) in Board::DIRECTIONS {
            let mut candidates = opponent & safe_shl(*friend & !guard, dir);
            while candidates != 0 {
                moves |= empty & safe_shl(candidates & !guard, dir);
                candidates = opponent & safe_shl(candidates & !guard, dir);
            }
        }
        Pieces { bits: moves }
    }
    /*
    ...
    */
}
```

`DIRECTIONS` is an array of tuples containg a direction and a guard. The direction is the value by which a set of bits should be left shifted to slide north, north-west, west, etc., and the guard is a set of bits which should never be shifted in that direction, lest they wrap around. For example, consider a bit on the left edge of the board (not the top left though). If this bit is left shifted by one, It will move one row up and onto the right edge of the board, not to the west. This is undesirable, so any bits about do be shifted in a given direction should be intersected (bitwise AND) with `!guard`. This is also explained in 2.2.3.

`safe_shl` is a closure (similar to a function, very similar to a lambda) allowing me to left shift by a negative number of bits, which the left shift operator does not allow in Rust. If I want to left shift `a` by `b` bits, I check if `b` is negative, and if so, I right shift `a` by `-b` bits for the desired behaviour. Closures are declared with pipe characters either side of their parameters.

`friend` is the bitboard containing the discs of the player about to move, and `opponent` is the bitboard with their opponent's discs.

`moves` is an unsigned 64-bit integer which I use to accumulate the legal moves.

For each `dir` (direction), I generate a set of candidates, which are squares which could be one square from a legal move in the direction currently being considered. If a candidate is one square away from an empty square in the current direction, that empty square is legal to move on, so I add it to the moves. I do this until there are no more candidates, and then start on the next direction until all the directions have been considered, at which point I return the moves, wrapped in a `Pieces` object (`struct Pieces` is explained in 3.1.2).

3.2.4 Implementation of `Board::make_move`

This method takes a mutable reference to a `Board` object, and a bit which is passed as an unsigned 64-bit integer (the integer should be of the form $2^k, 0 <= k <= 63$), which is the move to be made. It is not passed as an index, despite the fact that this would only use one byte instead of eight, because the caller of this method will usually have the move stored as an unsigned 64-bit integer and so having to convert the move back to an index at the call site, and back to a 64-bit integer in the function, is avoided.

First, the opponent's discs which will be flipped are computed. This is done by sliding a copy of the bit passed to the function in each direction until it is no longer at one of the opponent's discs, and flipping the discs it passes over if the last square it lands on is not one of the opponent's discs and not an empty square. Then comes the slightly cumbersome process of determing who should move next. As mentioned previously, if a player has no legal moves, it becomes their opponent's turn, and if the opponent also has no legal moves, the game ends in a win for the player with more discs of their color on the board. If there are an equal number of discs of each color on the board, the game ends in a draw. Because the legal moves have to be computed in order to determine who's turn it is after the move, the legal moves are returned because they are also often required by the caller, which avoids wasting time computing the legal moves twice.

The Code

```rust
impl Board {
    /*
    ...
    */
    pub fn make_move(&mut self, bit: u64) -> Pieces {
        // Returning the moves because the caller often needs it anyway
        let safe_shl = |a: u64, b: i8| if b > 0 { a << b } else { a >> (-b) };
        for (dir, guard) in Board::DIRECTIONS {
            let mut line = 0u64;
            let mut moving_bit = safe_shl(bit & !guard, dir);
            while moving_bit & self.waiting.bits != 0 {
                line |= moving_bit;
                moving_bit = safe_shl(moving_bit & !guard, dir);
            }
            if moving_bit & self.to_move.bits != 0 {
                self.to_move.bits |= line;
                self.waiting.bits &= !line;
            }
        }
        self.to_move.bits |= bit;
        std::mem::swap(&mut self.to_move, &mut self.waiting);
        self.black_moving ^= true;
        let mut moves = self.each_move();
        self.board_state = if moves.bits == 0 {
            std::mem::swap(&mut self.to_move, &mut self.waiting);
            self.black_moving ^= true;
            moves = self.each_move();
            if moves.bits == 0 {
                use std::cmp::Ordering;
                // No possible moves for both sides
                match self
                    .to_move
                    .clone()
                    .count()
                    .cmp(&self.waiting.clone().count())
                {
                    Ordering::Greater => BoardState::Won,
                    Ordering::Less => {
                        std::mem::swap(&mut self.to_move, &mut self.waiting);
                        self.black_moving ^= true;
                        BoardState::Won
                    }
                    Ordering::Equal => BoardState::Drawn,
                }
            } else {
                BoardState::Ongoing
            }
        } else {
            BoardState::Ongoing
        };
        moves
    }
    /*
    ...
    */
}
```

safe_shl is defined in the same way again to allow left shifting by a negative number of bits. Once the discs have been flipped, it becomes the opponent's turn with `self.black_moving ^= true;`. The ^ operator means XOR. The legal moves for the opposition are computed with `let mut moves = self.each_move();`, and if there are no legal moves for the opposition, it becomes the other player's turn again, and if that player has no legal moves either, the game is over and the winner is determined by counting the number of discs each player has.

3.2.5 Implementation of `Board::safe_make_move`

This method is very simple. Its parameters are the same as `Board::make_move`, but before making the move, the legality of the move is confirmed. The move is made in the same way as `Board::make_move` only if the move is legal. If the move is illegal, the move is not made.

The Code

```rust
impl Board {
    /*
    ...
    */
    pub fn safe_make_move(&mut self, bit: u64) -> Result<Pieces, String> {
        let mut moves = self.each_move();
        match moves.any(|x| x == bit) {
            true => {
                self.make_move(bit);
                Ok(moves)
            }
            false => Err(String::from("Move is not legal")),
        }
    }
    /*
    ...
    */
}
```

The return type `Result<Pieces, String>` is used because the function can fail, so the caller should be able to tell whether the function failed by checking which variant of the `Result` enum is returned. If the move is illegal, the string "Move is not legal" is returned, wrapped in the `Err` variant of the `Result` enum, otherwise, the legal moves are returned wrapped in the `Ok` variant of the `Result` enum. `moves.any|x| x == bit` evaluates to a boolean; true if any of the legal moves in the position are equal to the move (`bit`) passed to the function, and false if not.

3.2.6 Implementation of `Board::children`

This method takes a bitboard containing all the legal moves in a position, and returns a vector of tuples. The first item of the k^{th} tuple is the state of the board after having made the k^{th} legal move. The second item of each tuple is the set of legal moves in the position after having made the move. The set of legal moves for each child is returned because it must be computed anyway and very likely to be needed by the caller, so it would be a waste not to return it. The legal moves are passed in as an argument, despite being completely dependent on the board (the first argument), because the caller is likely to have computed them beforehand, so, again, this in order to avoid doing unnecessary computation.

The Code

```rust
impl Board {
    /*
    ...
    */
    pub fn children(&self, moves: &Pieces) -> Vec<(Board, Pieces)> {
        // Moves is a parameter as the caller will often already have calculated it
        // Returning both positions and the available moves as the latter
        // must be calculated and I don't want to waste the computation
        moves
            .clone()
            .map(|bit| {
                let mut next_board = self.clone();
                let next_board_moves = next_board.make_move(bit);
                (next_board, next_board_moves)
            })
            .collect()
    }
}
```

map is a function in Iterator's interface. It takes the implicitly passed object which implements Iterator and a closure, and returns a Map object. The fine details of this Map object are irrelevant, and at a basic level it is helpful to think of it as being synonymous with a Vec. In each element of the returned Map object is the value returned by the closure applied to the element at that position in the vector. collect is another function in Iterator's interface. In this context, it just converts the Map object into a Vec.

3.2.7 Implementation of frontier

frontier is a function which takes a Board object as its single argument, and returns a tuple containing two Pieces objects which store the locations of each of the frontier squares on the board for each player. A frontier square for a player is an empty square adjacent to at least one of that player's discs. The frontier squares for a player are computed by sliding that player's discs by one square for each direction, and intersecting with the empty squares.

The Code

```rust
fn frontier(board: &Board) -> (Pieces, Pieces) {
    let (mut to_move_front, mut waiting_front) = (0u64, 0u64);
    let empty = !(board.to_move.bits | board.waiting.bits);
    let safe_shl = |a: u64, b: i8| if b > 0 { a << b } else { a >> (-b) };
    // Slide in each direction
    for (dir, guard) in Board::DIRECTIONS {
        to_move_front |= safe_shl(board.to_move.bits & !guard, dir) & empty;
        waiting_front |= safe_shl(board.waiting.bits & !guard, dir) & empty;
    }
    (
        Pieces {
            bits: to_move_front,
        },
        Pieces {
            bits: waiting_front,
        },
    )
}
```

3.2.8 The Evaluation Function

The evaluation function is used to inexpensively evaluate how advantageous the current position is for either player, and is combined with an inexhaustive search of the game tree to find a good move within a reasonable length of time. It takes a bo The evaluation function I use calculates a linear combination of four important metrics: the difference in the number of discs in corners, the difference in the number of discs on the board, the difference in the number of frontier squares, and the number of available moves. When I say "the difference in" I mean how many more the current player has than their opponent. The evaluation function gives greater (more positive) evaluations for black, and more negative evaluations for white

The Code

```rust
pub fn better_eval(board: &Board, moves: &Pieces) -> i16 {
    let c = board.black_moving as i16 * 2 - 1;
    use BoardState::*;
    match &board.board_state {
        Won => i16::MAX * c,
        Drawn => 4 * c, // completely arbitrary
        Ongoing => {
            let corner_balance = c
                * (Pieces {
                    bits: board.to_move.bits & 0x8100000000000081,
                }
                .count() as i16
                    - Pieces {
                        bits: board.waiting.bits & 0x8100000000000081,
                    }
                    .count() as i16);
            let material_balance =
                c * (board.to_move.clone().count() as i16 - board.waiting.clone().count() as i16);
            let (moving_front, waiting_front) = frontier(board);
            let frontier_balance = c * (moving_front.count() as i16 - waiting_front.count() as i16);
            let move_count = c * moves.clone().count() as i16;
            corner_balance * 4000 + material_balance - frontier_balance * 10 + move_count * 1000
        }
    }
}
```

The variable declared on the first line of the function, `c`, is 1 if black is to move, and -1 if white is to move. This is a coefficient by which every metric is multiplied to make a negative score better for white, e.g., the `move_count` variable counts the number of moves available for the player who is about to move, so this should be multiplied by -1 if white is to move in order to keep negative scores better for white. The `match` statement checks whether the game is over yet, and if the game is won, the maximum or minimum score is returned depending on whether black or white won. If the game is drawn, `4 * c` is returned. It might seem strange that I did not give a draw an evaluation of zero, which is neither good nor bad for either color, but this evaluation function is not symmetric around zero like many others. This is because the move count is only calculated for the player to move and not for the player who is not about to move.

The greatest weight is given to the corners because corners are very important in Othello, as it is impossible to flip a disc placed on a corner. The number of discs on the corner squares each player has is computed by taking the bitwise AND of that player's pieces with the bits corresponding only to the corners, which in hexadecimal is `0x8100000000000081`.

3.2.9 Implementation of `minimax`

`minimax` is parameterized by an evaluation function which is used on leaf nodes, a board and the legal moves on the board (the caller, which is usually minimax because the algorithm is recursive, will have already computed the legal moves, so this is done in order not to waste that computation), a search depth which is the maximum number of moves ahead which minmax will search, and alpha and beta, which are used to prune parts of the search tree as discussed previously.

The Code

```rust
pub fn minimax<T: Ord + Clone + num::Bounded>(
    eval: fn(&Board, &Pieces) -> T, // Static evaluation function, although static is not enforceable
    (board, moves): &(Board, Pieces),
    depth: u8,    // How many moves ahead to we want to consider?
    mut alpha: T, // The evaluation of the best move found so far for black
    mut beta: T,  // alpha, but for white
) -> T {
    if depth == 0 || moves.bits == 0 {
        return eval(board, moves);
    }
    let children = board.children(moves);
    // This match statement avoids code duplication, where the same function is written almost identically for both colours
    let (mut best_eval, cmp) = match board.black_moving {
        true => (alpha.clone(), T::max as fn(T, T) -> T),
        false => (beta.clone(), T::min as fn(T, T) -> T),
    };
    for (next_board, next_moves) in children {
        best_eval = cmp(
            best_eval,
            minimax(
                eval,
                &(next_board, next_moves),
                depth - 1,
                alpha.clone(),
                beta.clone(),
            ),
        );
        if board.black_moving {
            alpha = best_eval.clone();
        } else {
            beta = best_eval.clone();
        }
        // Good positions for black give greater evaluations, opposite for white
        // Consider alpha and beta, the best evaluation for black so far and the best for white so far
        // If we compare the bes&mut t moves found so far for black and white and we find that the opposition
        // Can force a better position for them than the one we are currently considering, we need not consider
        // Any more children of this position as the opposition can and should avoid this position entirely
        // If they happen not to do that, that's all the better for us as they are playing sub-optimally
        if beta <= alpha {
            //println!("Prune!");
            break;
        }
    }
    best_eval
}
```

First, if the depth is zero or there are no more moves to make, then either no more positions should be searched or the game is over which implies that the node is a leaf node. This means that the evaluation function should be used, and the evaluation of that position returned. This is the base case of the minimax function.

In the recursive case, the child positions are generated using the `Board::children` method, whose implementation is explained in 3.2.6. The children are iterated through and the evaluation of each child position is calculated using a recursive call to minimax, with the `depth` argument as `depth - 1` and not `depth` because the call to minimax is made from one layer lower in the game tree. If `beta <= alpha`, no more positions need to be evaluated as it can be shown that this position will never be reached with perfect play as explained previously. `minimax` returns the best evaluation found.

3.2.10 Implementation of `best_move`

`minimax` only returns an evaluation of a position; it does not choose a move. `best_move` uses the evaluation returned by minimax to choose the best move. `best_move` is parameterized by an evaluation function, a `Board` object, and a search depth. The move which gives the position with the best evaluation is returned.

The Code

```rust
pub fn best_move<T: Ord + Clone + num::Bounded>(
    eval: fn(&Board, &Pieces) -> T,
    board: &Board,
    depth: u8,
) -> u64 {
    let moves = board.each_move();
    // Create a vector of tuples containing the evaluation of each child position
    and the move needed to reach that position
    let child_evals = board
        .children(&moves)
        .iter()
        .zip(moves)
        .map(|(pos, move_bit)| {
            (
                minimax(eval, pos, depth - 1, T::min_value(), T::max_value()),
                move_bit,
            )
        })
        .collect::<Vec<_>>();
    // Return the move corresponding to the greatest evaluation if black is to move
    , or the least evaluation if white is to move
    if board.black_moving {
        child_evals
            .iter()
            .max_by_key(|(value, _)| value)
            .expect("Moves should exist")
            .1
    } else {
        child_evals
            .iter()
            .min_by_key(|(value, _)| value)
            .expect("Moves should exist")
            .1
    }
}
```

3.3 Data Structure for the Graphical User Interface

There is one `struct` defined in the graphical user interface.

3.3.1 `Colorscheme`

I use this to store the colorscheme for the graphical user interface. When the user changes the colors of the graphical user interface, the fields of a `Colorscheme` object are adjusted accordingly.

The Code

```rust
#[derive(Copy, Clone, Debug)]
pub struct Colorscheme {
    board: (Color, Color),
    black: Color,
    white: Color,
    black_move: Color,
    white_move: Color,
}

impl Default for Colorscheme {
    fn default() -> Self {
        Colorscheme {
            board: (Color::from_hex(0x3a911au32), Color::from_hex(0x4ba30bu32)),
            black: Color::from_hex(0u32),
            white: Color::White,
            black_move: Color::from_hex(0x3f9e9bu32),
            white_move: Color::from_hex(0x3f9e9bu32),
        }
    }
}
```

`struct Colorscheme` is fairly self explanatory. The `board` field is a tuple containing two colors which are the colors of the squares on the board. There are two colors which allows the board to have a checkered pattern. `black` and `white` are the colors of the black and white pieces respectively (admittedly this is a misnomer). `black_move` and `white_move` fields are the colors used to draw the legal moves on the board. `black_move` corresponds to the blueish colored circles in 3.2. The `default` function returns a `Colorscheme` object which makes the board look like this:

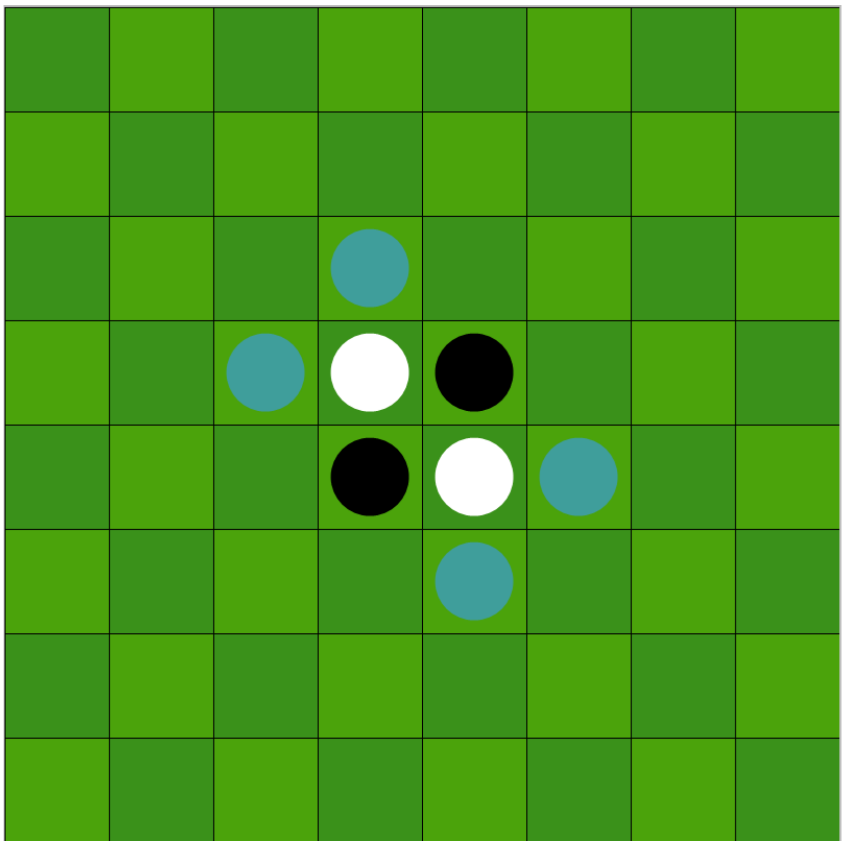


Figure 3.2: This is what the default colorscheme for my Othello GUI looks like. The blue circles are the legal moves for black.

3.4 Algorithms for the Graphical User Interface

Here are the functions defined for the graphical user interface. Some of them are quite long.

3.4.1 draw_board

This function (or as AQA would have it, procedure) takes coordinates `x: i32` and `y: i32`, a `side_len: i32` in pixels for the board to be drawn, a `piece_radius: i32` in pixels for the radius with which the discs should be drawn, a `board: &Board`, which is a read-only reference to a `Board` object representing an Othello position, and a `colorscheme: Colorscheme`, which determines which colors will be used to draw the board. It returns `()`, an empty tuple also called a unit type, because there is only one value of this type.

First, the squares of the Othello board are iterated through, and the appropriate colored square and circle is drawn at each point. Next, crisscrossed lines are drawn on the board to make the borders of each square clear.

The Code

```rust
fn draw_board(
    x: i32,
    y: i32,
    side_len: i32,
    piece_radius: i32,
    board: &Board,
    colorscheme: Colorscheme,
) {
    let moves = board.each_move();
    let sqsize = side_len / 8;
    let (moving_color, waiting_color, move_color) = match board.black_moving {
        true => (colorscheme.black, colorscheme.white, colorscheme.black_move),
        false => (colorscheme.white, colorscheme.black, colorscheme.white_move),
    };

    for i in 0..8 {
        for j in 0..8 {
            draw::draw_rect_fill(
                i * sqsize + x,
                j * sqsize + y,
                sqsize,
                sqsize,
                if (i + j) % 2 == 0 {
                    colorscheme.board.0
                } else {
                    colorscheme.board.1
                },
            );
            let cur_bit = 1 << (i * 8 + j);

            // Check if we should show a piece or possible move on this square
            if (board.to_move.bits | board.waiting.bits | moves.bits) & cur_bit ==
    0 {
                continue;
            }
            draw::set_draw_color(if board.to_move.bits & cur_bit != 0 {
                moving_color
            } else if board.waiting.bits & cur_bit != 0 {
                waiting_color
            } else {
                move_color
            });
            // We have chosen the color already, so now draw the circle
            draw::draw_pie(
                i * sqsize + sqsize / 2 - piece_radius + x,
                j * sqsize + sqsize / 2 - piece_radius + y,
                2 * piece_radius,
                2 * piece_radius,
                0.0,
                360.0,
            );
        }
    }

    draw::set_draw_color(Color::Black);
    for i in 0..=8 {
        draw::draw_line(i * sqsize + x, y, i * sqsize + x, 8 * sqsize + y);
        draw::draw_line(x, i * sqsize + y, 8 * sqsize + x, i * sqsize + y);
    }
}
```

This function is long, but the bulk of it is made up of conditionals and coordinate geometry spread over multiple lines. I will not fully explain all of the coordinate geometry, but the factors of eight which appear throughout this function are always as a consequence of Othello being played on an eight by eight board.

3.4.2 main

This function tells FLTK to construct a window containing an Othello board and the score for black and white, and a button to change the size of the discs. The window is shown in figure 3.3.

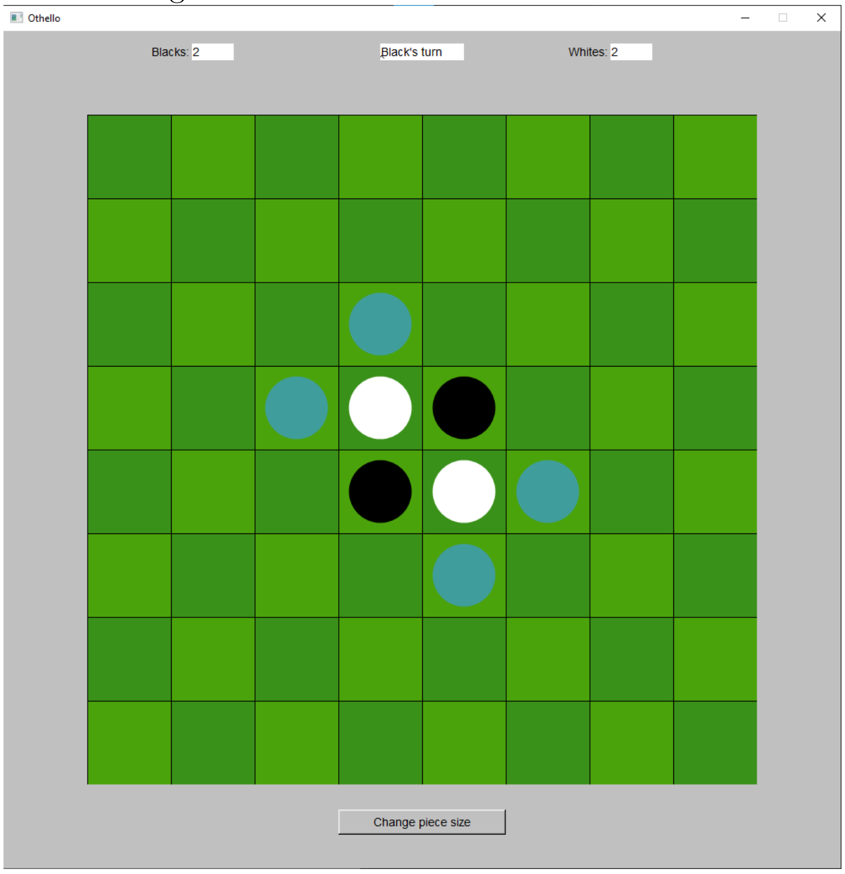


Figure 3.3: This is what the window looks like when you are playing Othello.

`main` takes four arguments: `app: app::App` which is an object needed to make certain calls to FLTK, `human_black: bool` which is a boolean determing which color the human will play, `depth: u8` which is the depth at which minimax will be run to find good moves for the computer player, and `colorscheme: Colorscheme`, which is passed into `draw_board` whenever the board needs to be drawn. `main` is called when the player clicks a button from the main menu, creating a new window which the player will use to play against the computer.

The top three labels show how many pieces black has, whose turn it is, and how many pieces white has.

The Code

I will list the code for `main` in several snippets because the function is over 150 lines in total.

Some of the variables are wrapped in an `Arc` object. `Arc` is a thread-safe reference-counting pointer. This allows me to safely capture objects in closures, possibly across different threads, by copying their pointer, so that they point to the same underlying data. Were this pointer not reference-counting, the number of pointers pointing to the same data would not be known, and if there were eventually zero pointers pointing to the data, the memory which the data occupied would not be freed despite being inaccessible, causing a memory leak.

Some of the variables are wrapped in a `Mutex` object. `Mutex` allows a variable to be shared and mutated between threads with a locking mechanism. Only one thread can hold a `Mutex` lock at any one time, and having the lock allows that thread to mutate the data within the `Mutex` while no other thread can.

```
pub fn main(app: app::App, human_black: bool, depth: u8, colorscheme: Colorscheme)
    {
    // Create the sender and reciever
    let (tx, rx) = mpsc::channel::<u64>();
    // Create board and window
    let board = Arc::new(Mutex::new(Board::default()));
    let mut wind = Window::new(700, 40, 1000, 1000, "Othello");
    // Create textbox showing whose move it is
    let mut to_move = Output::new((wind.width() - 160) / 2, 15, 160, 20, "");
    to_move.set_frame(FrameType::FlatBox);
    // Create textboxes showing how many discs black and white each have
    let (mut blacks, mut whites) = (
        Output::new((wind.width() - 100) / 4, 15, 50, 20, "Blacks:"),
        Output::new((wind.width() * 3 - 100) / 4, 15, 50, 20, "Whites:"),
    );
    blacks.set_frame(FrameType::FlatBox);
    whites.set_frame(FrameType::FlatBox);
    // Where the board will be drawn
    let mut frame = Frame::new(
        (wind.width() - SIDE_LEN) / 2,
        (wind.height() - SIDE_LEN) / 2,
        SIDE_LEN,
        SIDE_LEN,
        "",
    );
    frame.set_color(Color::Blue);
    let surf = Arc::new(ImageSurface::new(frame.width(), frame.height(), false));
    /*
        ...
    */
}
```

First, a sender and receiver (`tx` and `rx` respectively) are created. These are used to send information between threads, as this is a multithreaded program. Next, the `Board` object is created using the `default` function explained in 3.2.1. In this function, the `board` variable is actually a `Board` wrapped in a `Mutex` wrapped in an `Arc`. The `Window` object (an abstraction in the FLTK library over a display manager's window) is created. The text boxes showing the number of pieces black has, whose move it is, and how many pieces white has, are created. In FLTK, these have type `Output`. Next, the canvas (called a `Frame` in FLTK) onto which the board will be drawn is created. Next, the surface (called an `ImageSurface`) onto which the board will be drawn is created.

```
pub fn main(app: app::App, human_black: bool, depth: u8, colorscheme: Colorscheme)
    {
    /*
        ...
    */
    let piece_radius = Rc::new(Cell::new(SIDE_LEN * 3 / 64));
    frame.draw({
        let piece_radius = piece_radius.clone();
        let board = board.clone();
        let surf = surf.clone();
        move |fr| {
            ImageSurface::push_current(&surf);
            draw_board(
                0,
                0,
                SIDE_LEN,
                piece_radius.get(),
                &board.lock().unwrap(),
                colorscheme.clone(),
            );
            ImageSurface::pop_current();
            surf.image().unwrap().draw(fr.x(), fr.y(), fr.w(), fr.h());
        }
    });
    /*
        ...
    */
}
```

The first line of this snippet sets, `piece_radius`, which is the radius with which the pieces (discs) will be drawn. Letting L be the length of the side of the board and r the radius of the pieces,

$$r = \frac{3L}{64} = \frac{L}{8} \times \frac{3}{8} \implies \frac{d}{2} = \frac{3l}{8} \implies d = \frac{3l}{4},$$

where d is the diameter of the pieces, and l is the length of the side of a square on the board. In other words, the diameter of each disc is three quarters of the length of the side of each square. `piece_radius` is wrapped in a `Cell` which allows multiple handles to `piece_radius` to exist, all of which can safely mutate `piece_radius`, or rather to mutate the value held within the `Cell` within the `Rc` which is `piece_radius`. A `Rc` is similar to an `Arc` but cannot be shared between threads, which gives it a lower runtime cost as fewer runtime checks need to be made.

The rest of this snippet sets the draw callback for the `frame` (the `frame` is the object containing the surface onto which the board will be drawn). A callback is just another name for a closure (like a lambda). The closure is called whenever the `frame` redraws itself.

```
pub fn main(app: app::App, human_black: bool, depth: u8, colorscheme: Colorscheme)
    {
    /*
        ...
    */
    frame.handle({
        let board = board.clone();
        move |frame, event| match event {
            Event::Push => {
                let board = board.lock().unwrap().clone();
                if event_button() == 1 && human_black == board.black_moving {
                    let (x, y) = event_coords();
                    let (x, y) = (
                        std::cmp::min(8 * (x - frame.x()) / SIDE_LEN, 7),
                        std::cmp::min(8 * (y - frame.y()) / SIDE_LEN, 7),
                    );
                    let move_bit = 1 << (8 * x + y);
                    tx.send(move_bit).expect("Receiver hung up");
                };
                true
            }
            _ => false,
        }
    });
    /*
        ...
    */
}
```

The above part of the function sets the handle closure for the frame, which allows clicks on the board to be registered. When the player clicks on the board, an `Event::Push` is passed to closure along with the frame object itself. A move is only sent to the thread handling the moves (as well as the computer player - the creation of this thread comes later in the function) if the click was a left-click and if it is the human's turn. Even if this is the case, the thread handling the moves calls the `Board::safe_make_move` function explained in 3.2.5, which does (almost) nothing if the move passed to it is illegal.

The move is computed from the relative coordinates of the click, `x` and `y`, in the line `let move_bit = 1 << (8 * x + y)`. The reason for the expression `8 * x + y` is explained in ??.

```rust
pub fn main(app: app::App, human_black: bool, depth: u8, colorscheme: Colorscheme)
    {
    /*
        ...
    */
    let frame = Arc::new(Mutex::new(frame));
    let mut button = Button::new(
        wind.width() / 2 - 100,
        SIDE_LEN + 30 + (*frame).lock().unwrap().y(),
        200,
        30,
        "Change piece size",
    );
    button.set_callback({
        let piece_radius = piece_radius.clone();
        let frame = frame.clone();
        move |_but| {
            piece_radius.set(SIDE_LEN * 5 / 64 - piece_radius.get());
            app::lock().expect("Locking unsupported");
            let mut frame = frame.lock().unwrap();
            frame.redraw();
            app::unlock();
            drop(frame);
        }
    });
    /*
        ...
    */
}
```

The above part of the function firstly wraps the frame in a `Mutex` wrapped in an `Arc` to allow it to be shared between threads. The frame needs to be shared between threads in order for the board to be redrawn the instant a move is made from another thread, otherwise the board will freeze while the computer is making moves (remember, if one player has no legal moves but the other player does, that player makes moves until the first player has a legal move to make).

Next, the button allowing the size of the discs to be changed at runtime is created. The arguments passed to `Button::new` determine the text shown on the button, the dimensions of the button, and the location of the button. Next, button's callback is set. This is called whenever the button is clicked. It changes the size of the discs on the board. Because the frame is wrapped in a `Mutex`, the frame object itself must first be acquired by locking the mutex before `Frame::redraw` can be called.

```rust
pub fn main(app: app::App, human_black: bool, depth: u8, colorscheme: Colorscheme)
    {
    /*
        ...
    */
    thread::spawn({
        let frame = frame.clone();
        let board = board.clone();
        move || {
            let update = |whites: &mut Output,
                          blacks: &mut Output,
                          to_move: &mut Output,
                          board: &Board| match board.black_moving {
```

```
                true => {
                    whites.set_value(&board.waiting.count().to_string());
                    blacks.set_value(&board.to_move.count().to_string());
                    to_move.set_value("Black's turn");
                }
                false => {
                    blacks.set_value(&board.waiting.count().to_string());
                    whites.set_value(&board.to_move.count().to_string());
                    to_move.set_value("White's turn");
                }
            };
            /*
                ...
            */
        }
    /*
        ...
    */
}
```

This is the first part of the closure which the separate thread runs. It declares a closure called `update` which takes the output boxes which display the number of pieces white and black have and whose move it is, as well as a reference to a clone of the board, and updates the text in the output boxes to reflect a change in the number of pieces each player has on the board, and whose move it is.

```

pub fn main(app: app::App, human_black: bool, depth: u8, colorscheme: Colorscheme) {
    /*
        ...
    */
            loop {
                // Store a copy of the board instead of the lock itself so other threads can access it while this thread computes a move
                let mut board_clone = board.lock().unwrap().clone();
                while board_clone.black_moving != human_black
                    && board_clone.board_state == BoardState::Ongoing // While it is the computer's turn
                {
                    update(&mut whites, &mut blacks, &mut to_move, &board_clone);
                    let move_bit =
                        evaluation::best_move(evaluation::better_eval, &board_clone, depth);
                    board_clone.make_move(move_bit);
                    board.lock().unwrap().make_move(move_bit);
                        // Stop other threads from accessing the GUI while this thread is doing so
                        app::lock().expect("Locking unsupported");

                        let mut frame = frame.lock().unwrap();
                        frame.redraw();
                        update(&mut whites, &mut blacks, &mut to_move, &board_clone);
                        app::unlock();

                        app::awake();
                }
                update(&mut whites, &mut blacks, &mut to_move, &board_clone);
                let board_state = board.lock().unwrap().board_state;
                match board_state {
                    BoardState::Ongoing => {
                        let move_bit = rx.recv().unwrap();
                        board.lock().unwrap().safe_make_move(move_bit);
                    }
                    BoardState::Drawn => {
                        to_move.set_value("Game ended in a draw.");
                        return;
                    }
                    BoardState::Won => {
                        let winning_color = match board.lock().unwrap().
    black_moving {
                            true => "Black",
                            false => "White",
                        };
                        let msg = format!("{winning_color} wins!");
                        to_move.set_value(&msg);
                        return;
                    }
                }
                app::lock().expect("Locking unsupported");
                frame.lock().unwrap().redraw();

                app::unlock();
                app::awake();
            }
        }
    });
    wind.end();
    wind.show();
    app.run().unwrap();
}
```

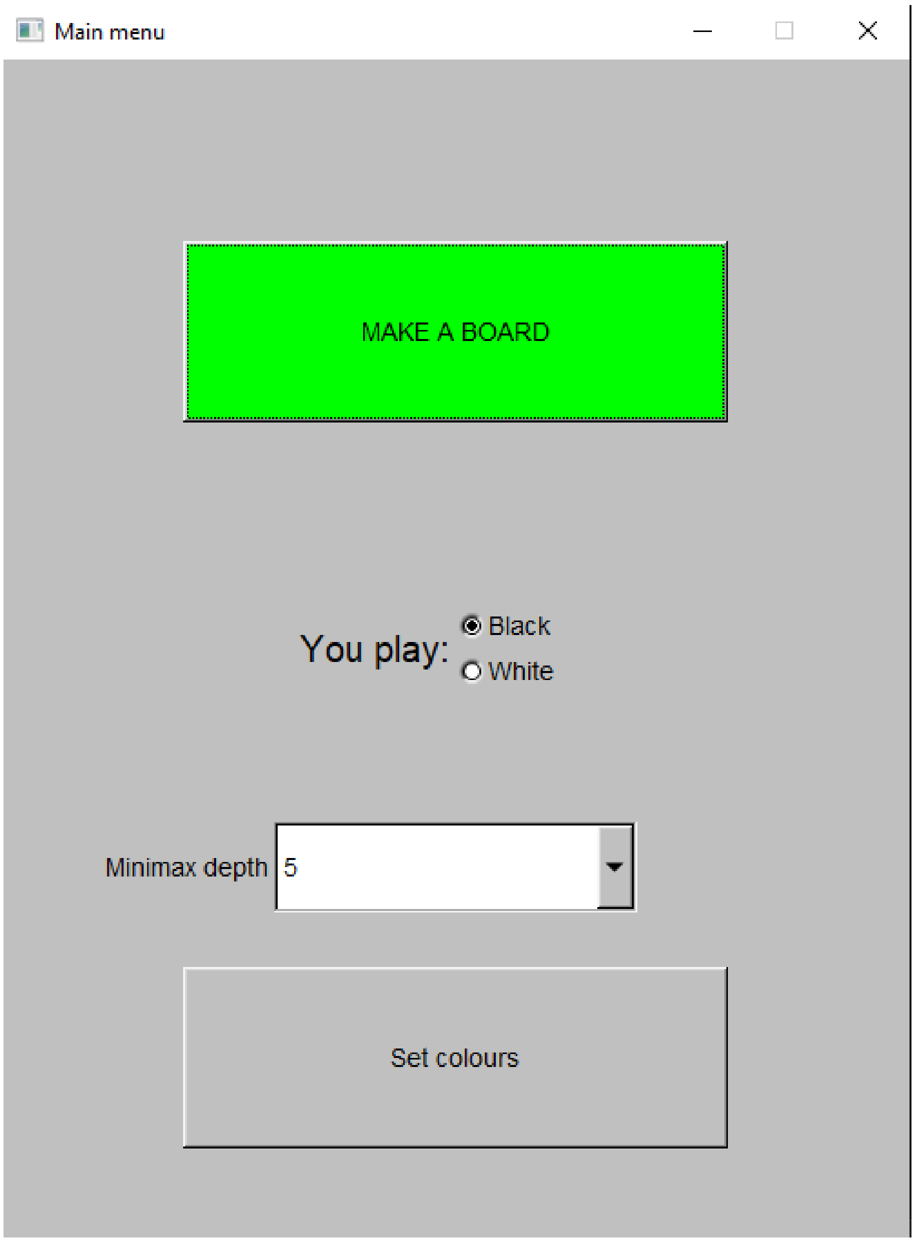
In the snippet above, the code enclosed by the `loop` is repeated until the Othello window is closed. First, a clone of the board is acquired. This is done instead of acquiring a lock on the board (which would be held until it dropped out of scope) in order to minimize the length of time for which this thread locks the board, because minimax can take a long time to run (depending on the depth) and I do not want the main thread to have to wait for a lock on the board for a considerable length of the time, which could make the program slow to respond. Next, the computer finds (`let move_bit = evaluation::best_move(...);`) makes moves (`board.lock().unwrap().make_move(move_bit);`) while it is the computer's turn
(`board_clone.black_moving != human_black` i.e., the human is not moving $\implies$ the computer is moving) and the game is ongoing (`board_clone.board_state == BoardState::Ongoing`). The board is redrawn (`frame.redraw()`) to show the move immediately.

Next, the `blacks`, `whites`, and `to_move` output boxes are updated. If the game is finished, the closure returns and stops looping, otherwise the thread waits until another move is sent and makes more moves. The frame is redrawn one last time.

`wind.end();`, `wind.show();`, and `app.run().unwrap();` tell FLTK that I am done adding widgets to the window, and to show the window on the screen.

3.4.3 `board_setup`

This function builds the window containing various options for the creation of the board. This window is shown in figure 3.4.



Minimax depth 5

Figure 3.4: This is the window containing the options for setting up a game of Othello with the computer.

This window is the first and only to be launched as soon as the program runs.

The Code

```
pub fn board_setup() {
    let app = app::App::default();
    // Make the main menu window
    let mut wind = Window::new(300, 300, 500, 700, "Board Setup");
    // Button for starting an Othello game
    let mut but = Button::new(100, 100, 300, 100, "PLAY");
    // This will hold the radio buttons through which the user will select the
   color which they play
    let mut choices = group::Group::new(250, 300, 100, 50, Some("You play:"));
    let colorscheme = Rc::new(Cell::new(Colorscheme::default()));
    choices.set_align(fltk::enums::Align::Left);
    choices.set_label_size(20);
    let mut choice = RadioRoundButton::new(250, 300, 100, 25, "Black");
    choice.toggle(true);
    choices.add(&choice);
    choices.add(&RadioRoundButton::new(250, 325, 100, 25, "White"));
    choices.end();

    let mut depth = Choice::new(150, 420, 200, 50, "Minimax depth");
    (1..10).for_each(|n| depth.add_choice(&n.to_string()));
    depth.set_value(4);

    but.set_callback({
        let app = app.clone();
        let colorscheme = colorscheme.clone();
        move |_| {
            main(
                app.clone(),
                choice.value(),
                depth.value() as u8 + 1u8,
                colorscheme.get(),
            );
        }
    });
    but.set_color(Color::Green);

    let mut colortrig = Button::new(100, 500, 300, 100, "Set colors");
    colortrig.set_callback({
        let app = app.clone();
        let colorscheme = colorscheme.clone();
        move |_| colorscheme_setup(app.clone(), colorscheme.clone())
    });

    wind.end();
    wind.show();
    app.run().unwrap();
}
```

All this function does is construct some buttons and dropdown menus inside a window, placed at the appropriate coordinates on the window. It sets the callback for `but` as `main`, passing the arguments which the player has chosen, e.g. which color they play. The callback for `colortrig` is set to `colorscheme_setup`, which I will get to in 3.4.4, but just briefly, `colorscheme_setup` makes a window with color pickers allowing the player to customize the colors of the board to their liking.

3.4.4 `colorscheme_setup`

This function takes an FLTK `App` object and a `Colorscheme` wrapped in a `Cell Rc`. `Cell` is a type from Rust's standard library implementing interior mutability, which allows me to mutate the `Colorscheme` object from this function while having a reference to it in `board_setup`. I need to mutate the `Colorscheme` object whenever the user applies a change to the colorscheme. `Rc` is a reference-counting pointer, which allows me to have multiple handles pointing to the same data within a program (one in the `board_setup` function, one in the `colorscheme_setup` function).

In this function, a color-picker widgets are created on top of a window, which allow the user to control the colors of the `Colorscheme` object passed into this function. A screenshot of the window is shown in figure 3.5.

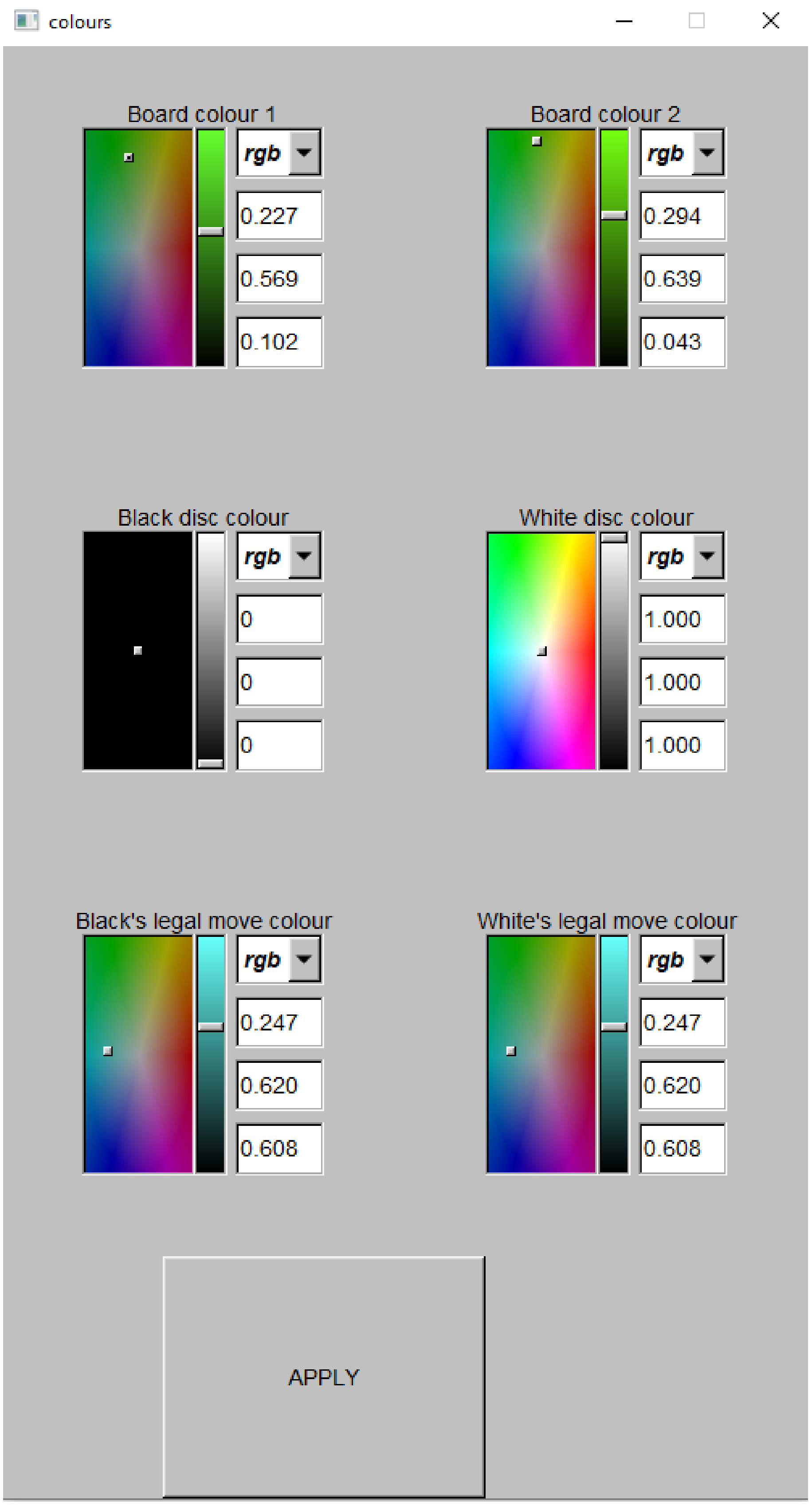


Figure 3.5: This window allows the user to change the colorscheme of the Othello board.

The Code

```rust
fn colorscheme_setup(app: app::App, colorscheme: Rc<Cell<Colorscheme>>) {
    let mut wind = Window::new(300, 300, 500, 900, "colors");
    // Make each color picker
    let mut boardcolors = (
        ColorChooser::new(50, 50, 150, 150, "Board color 1"),
        ColorChooser::new(300, 50, 150, 150, "Board color 2"),
    );
    let mut disccolors = (
        ColorChooser::new(50, 300, 150, 150, "Black disc color"),
        ColorChooser::new(300, 300, 150, 150, "White disc color"),
    );
    let mut movecolors = (
        ColorChooser::new(50, 550, 150, 150, "Black's legal move color"),
        ColorChooser::new(300, 550, 150, 150, "White's legal move color"),
    );
    boardcolors
        .0
        .set_tuple_rgb(colorscheme.get().board.0.to_rgb());
    boardcolors
        .1
        .set_tuple_rgb(colorscheme.get().board.1.to_rgb());
    disccolors
        .0
        .set_tuple_rgb(colorscheme.get().black.to_rgb());
    disccolors
        .1
        .set_tuple_rgb(colorscheme.get().white.to_rgb());
    movecolors
        .0
        .set_tuple_rgb(colorscheme.get().black_move.to_rgb());
    movecolors
        .1
        .set_tuple_rgb(colorscheme.get().white_move.to_rgb());

    let mut apply = Button::new(100, 750, 200, 150, "APPLY");
    // When apply is clicked, change the colors of the colorscheme object
    apply.set_callback({
        let colorscheme = colorscheme.clone();
        move |_| {
            let next = Colorscheme {
                board: (
                    Color::from_hex(boardcolors.0.hex_color()),
                    Color::from_hex(boardcolors.1.hex_color()),
                ),
                black: Color::from_hex(disccolors.0.hex_color()),
                white: Color::from_hex(disccolors.1.hex_color()),
                black_move: Color::from_hex(movecolors.0.hex_color()),
                white_move: Color::from_hex(movecolors.1.hex_color()),
            };
            colorscheme.set(next);
        }
    });
    wind.end();
    wind.show();
    app.run().unwrap();
}
```

`ColorChooser` is a color-picker widget built into FLTK, four of which are shown in figure 3.5. The player can choose the colors by dragging the white square onto the desired color. When the user presses the apply button, the closure passed into `apply.set_callback` is called, which changes the values in the colorscheme which was originally passed to the function to the values specified by the player. This will affect the colorscheme of any new games the player starts.

Chapter 4

Testing

4.1 Tests for Fulfillment of Objectives

In this section, there is a test for each of the objectives in 1.4.2.

4.1.1 Objective One

Objective one: *Upon launching the program, a graphical user interface should be shown to the user.*

Input

This is easy to test. I will run the program by entering `cargo run` into the command line. Cargo is a program provided by Rust. `cargo run` recompiles the code if there is any source file which was modified later than the executable was created. The executable resulting from compiling the code is run.

Expected Outcome

I expect a window like the one in figure 3.4 to appear on the screen.

Outcome

The outcome is exactly as expected. The evidence is in figure 3.4. I have also demonstrated it in my video (`https://youtu.be/RsrDGrH2LLQ?t=7`).

4.1.2 Objective Two

Objective two: *When playing Othello, a disc should be placed on the board when a square is clicked if and only if it is the human players turn, and the move which they are making is legal.*

Input

The Setup: I will run the program, and a window just like the one shown in figure 3.4 should appear. I will adjust the depth to nine in order to make the computer player take a long time to make a move which will allow me to check that the human cannot make moves when it not the human's turn. I will click the button saying *MAKE A BOARD* shown in figure 3.4, and the Othello board window like the one shown in figure 3.3 should appear.

The Test: I will click several squares on the board which are *not* highlighted as legal moves. I will click on a square which is highlighted as a legal move. I will play until the position becomes more complicated which will make the computer player take longer to compute each move. I will click on a square which I know would be a legal move for the human if it were the human's turn when it is not the human's turn.

Expected Outcome

When I click any squares which are not highlighted as (and are not) legal moves, the state of the board should not change. When I click on a square which is highlighted as (and is) a legal move, a disc should be placed on the square which was clicked on. When I click on any square on the board while it is the computer player's turn, the state of the board should not change.

Outcome

This test works exactly as expected as shown in my video (`https://youtu.be/RsrDGrH2LLQ?t=33`).

4.1.3 Objective Three

Objective three: *The difficulty of the computer player must be selectable through the graphical user interface.*

Input

I will run the program and click on the dropdown menu labeled *Minimax depth* on the main menu shown in figure 3.4. I will choose a depth of one first. I will play one game against the computer player. I will choose a depth of eight from the main menu, and play one game against the computer player again.

Expected Outcome

I should be able to beat the computer player easily at minimax depth one, and the computer should make moves with imperceptible delay. The computer player should beat me easily at minimax depth eight, and the computer should make moves with a noticeable delay because the number of positions which need to be evaluated grows exponentially with the depth.

Outcome

I was able to beat the computer player at a depth of one, and the computer player's moves were made almost instantly. I lost to the computer player at a depth of eight, and the computer player's moves took over a second during the middle of the game, but were very fast at the beginning and end of the game. This outcome is as expected. I have demonstrated this in my video (`https://youtu.be/RsrDGrH2LLQ?t=72`).

4.1.4 Objective Four

Objective four: *Whether the human plays white or black must be selectable through the graphical user interface.*

Input

I will run the program and start a game of Othello. I will change the color of the human from black to white through the main menu shown in figure 3.4 and start another game of Othello.

Expected outcome

In the first game the human should control the black discs because the human plays black by default. In the second game the human should control the white discs.

Outcome

This test runs just as expected. The radio buttons allow the human to switch between playing black and white work. I have demonstrated this in my video (`https://youtu.be/RsrDGrH2LLQ?t=114`).

4.1.5 Objective Five

Objective five: *The user must be able to choose the color of the discs and squares on the board with a color picker.*

Input

I will run the program and click the button reading *Set colors* from the main menu shown in figure 3.4. I will then adjust the colors for each part of the board using their associated color pickers and click *APPLY*. I will start a game of Othello from the main menu.

Expected Outcome

Firstly, when I click the button reading *Set colors*, a new window like the one in figure 3.5 should open. When I adjust the colors arbitrarily and press *APPLY*, any new games of Othello started should have the colorscheme which I chose.

Outcome

This test runs exactly as expected so it is passed. An example of the colors which can be chosen for the board is shown in figure 4.1. I have also demonstrated this in my video (`https://youtu.be/RsrDGrH2LLQ?t=356`).

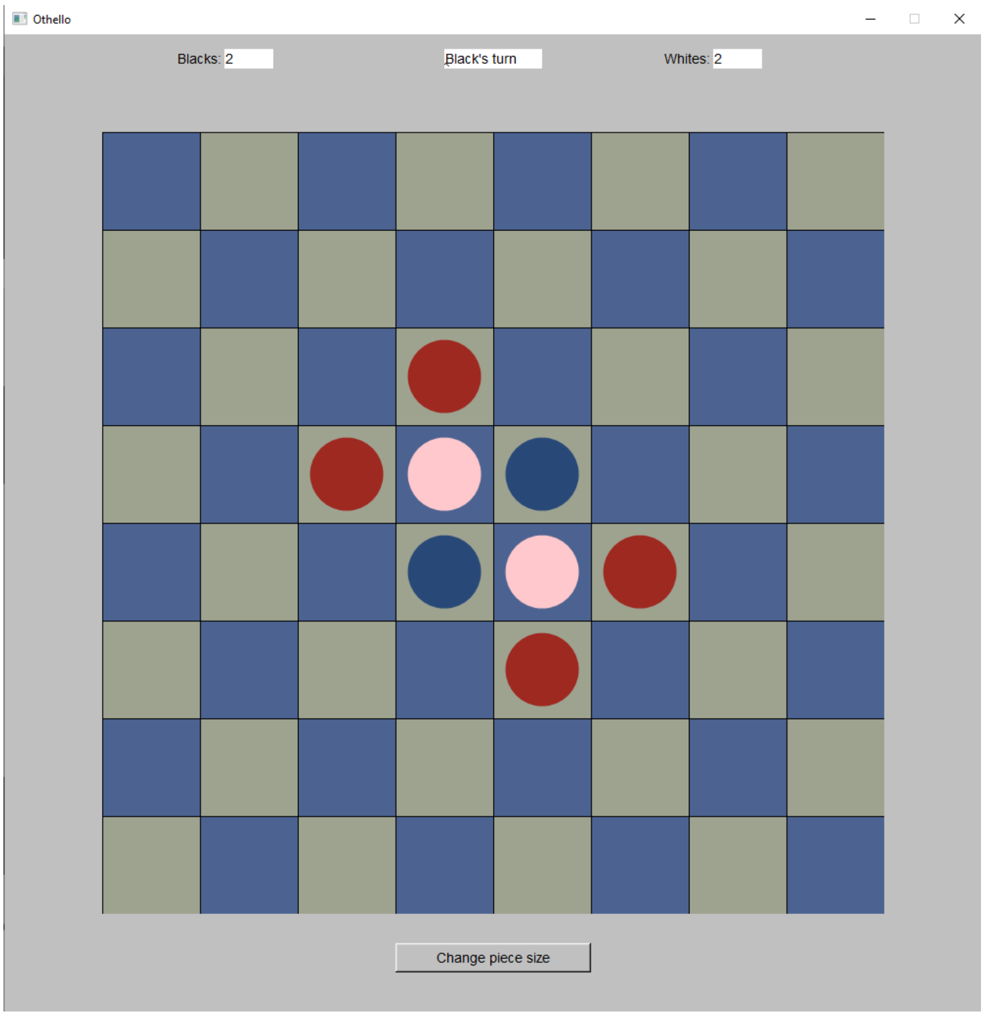


Figure 4.1: These colors have been chosen arbitrarily and show that the colors of the board are completely customizable.

4.1.6 Objective Six

Objective six: *The computer player must make moves good enough to beat the end user, taking no longer than five seconds to make one move.*

Input

Ask my end user to play against the computer player at a minimax depth of seven.

Expected Outcome

The computer player should win, taking no longer than five seconds for each move.

Outcome

The computer player won and took no longer than five seconds for each move. This test has been passed.

4.1.7 Objective Seven

Objective seven: *The graphical user interface should display the number of discs belonging to each player on the board.*

Input

Run the program and start a game of Othello. Play against the computer for a few moves, and after each move, count the number of black discs and white discs on the board and compare the counted values with the values shown at the top of the window, like the one shown in figure 3.3.

Expected Outcome

The counted values should always be equal to the values shown at the top of the window.

Outcome

This outcome of this test was just as expected, so this test passes. I have demonstrated this in my video (`https://youtu.be/RsrDGrH2LLQ?t=162`).

4.1.8 Objective Eight

Objective eight: *When the game is over, the graphical user interface should show whether the game was won or drawn, and if the game was won, which player won.*

Input

Run the program and start a game of Othello. Play against the computer until the game is over. Count the number of discs each player has on the board, and check who the graphical user interface says has won or if the game has ended in a draw.

Expected Outcome

If the number of discs for each player is equal, the graphical user interface should say that the game ended in a draw. If black has more discs than white, the graphical user interface should say that black has won the game. Otherwise, the graphical user interface should say that white has won the game.

Outcome

This test runs exactly as expected. In figure 4.2, white has won and at the top of the window there is some text reading *White wins!*, agreeing with the outcome of the game. I have also demonstrated this in my video (`https://youtu.be/RsrDGrH2LLQ?t=198`).

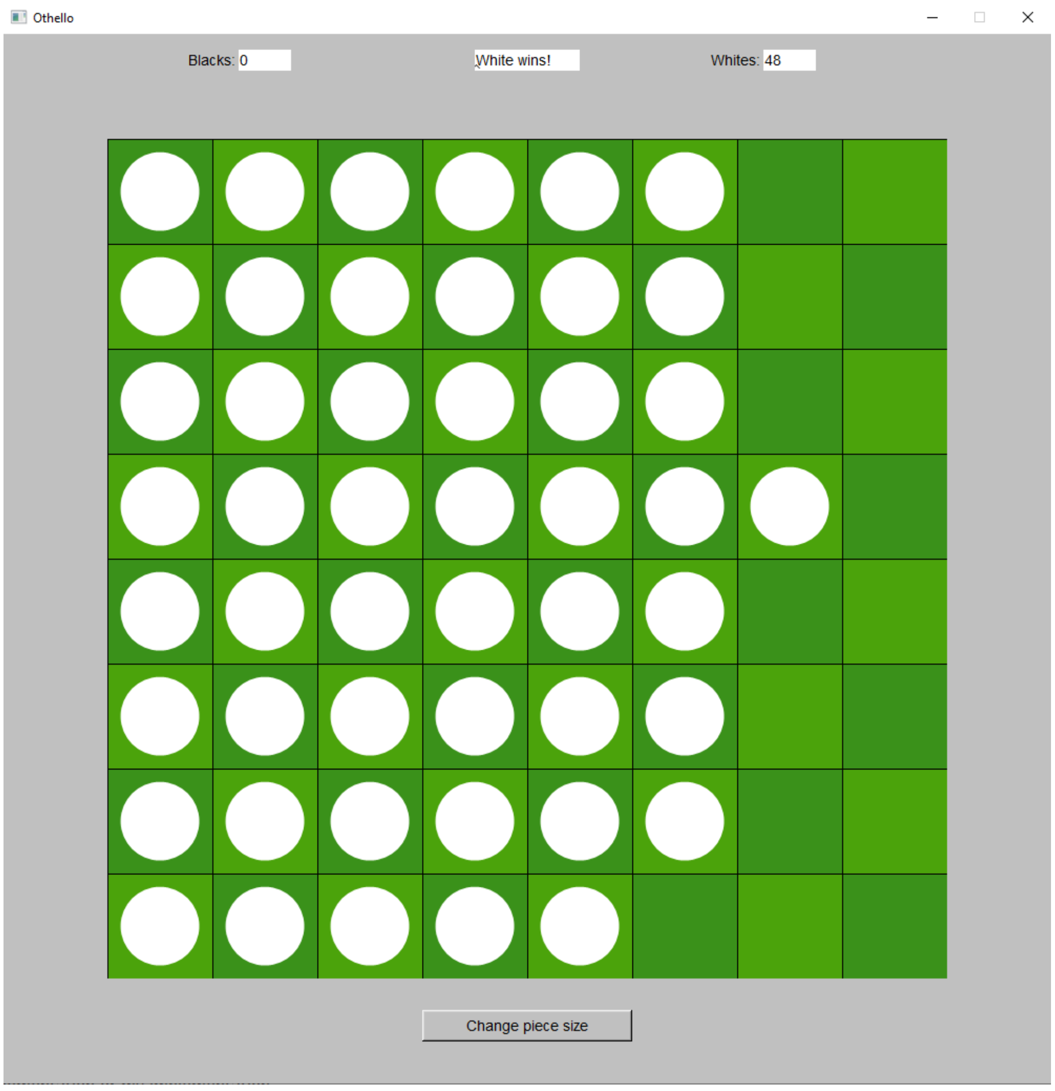


Figure 4.2: In this position, white has clearly won, and the GUI agrees.

4.1.9 Objective Nine

Objective nine: *The graphical user interface must remain responsive while the computer player is computing a move.*

Input

I will run the program and select a minimax depth of nine from the main menu shown in 3.4. I will play against the computer player until it takes enough time to make a move for me to click the *Change piece size* button at the bottom of the Othello window while the computer player is figuring out a move, like the one in 3.3.

Expected Outcome

The size of the discs on the board should change immediately just as they do when the same button is clicked when the computer player is not making a move.

Outcome

This test passes as expected and I have demonstrated this in my video (`https://youtu.be/RsrDGrH2LLQ?t=249`).

4.1.10 Objective Ten

Objective ten: *The user must be able to toggle between small and large discs when playing Othello.*

Input

I will run the program and start a game of Othello. I will click the button reading *Change disc size* twice.

Expected Outcome

I expect the size of the discs to get smaller the first time the button is pressed, and return to their original size the second time the button is pressed.

Outcome

This test passes as expected. The appearance of the board after having pressed the *Change disc size* button once is shown in figure 4.3. I have also demonstrated this in my video (`https://youtu.be/RsrDGrH2LLQ?t=309`).

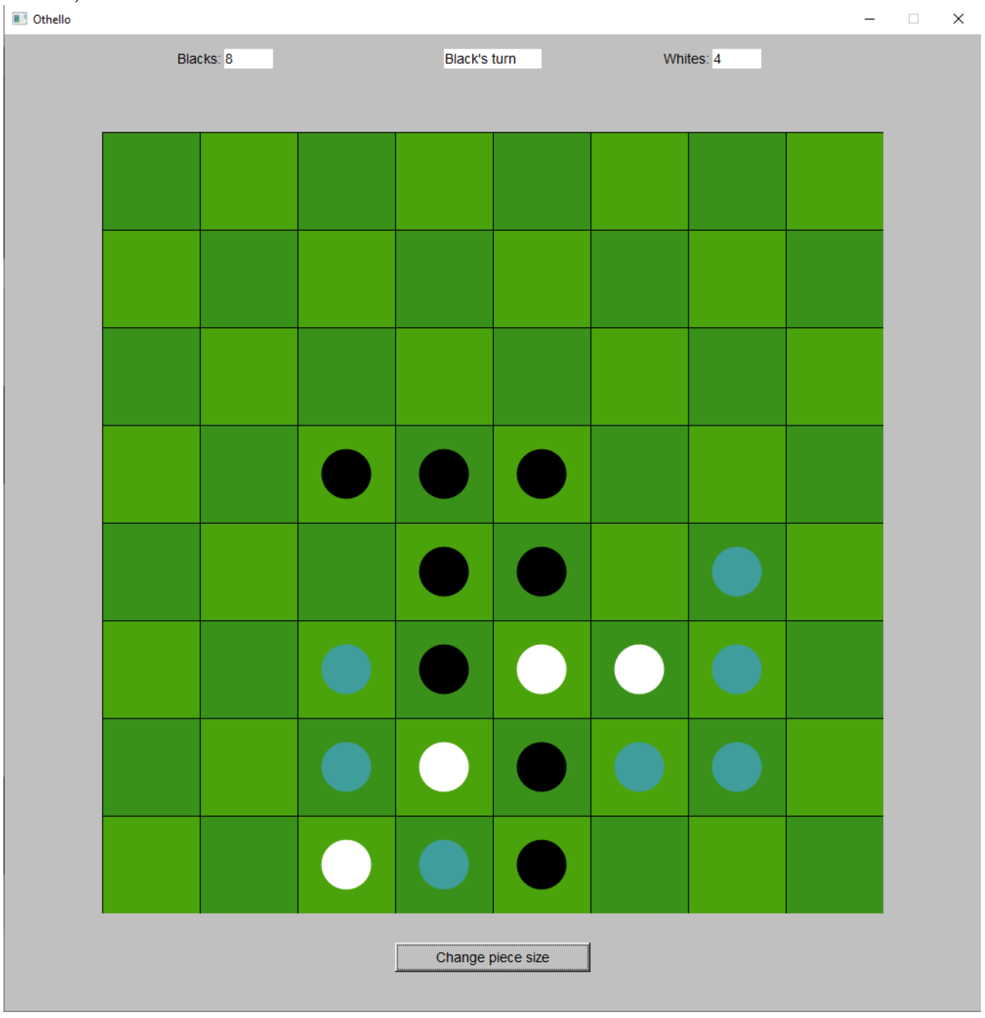


Figure 4.3: This is what the Othello board looks like with the discs toggled to their smaller size.

4.1.11 Objective Eleven

Objective eleven: *The legal moves of the player whose move it is must be shown by the graphical user interface on the board.*

Input

I will run the program and select a minimax depth of nine. I will start a game of Othello, and after every move, look at which moves are declared to be legal moves by the graphical user interface. I will make sure that these moves are legal and that no legal moves are not declared to be legal by figuring out for myself what the legal moves are in each position.

Expected Outcome

I expect the moves which the graphical user interface declares to be legal and the moves which are actually legal to be identical.

Outcome

This test passes as expected. Some evidence of this is in figure 3.3 and I have demonstrated it in my video (`https://youtu.be/RsrDGrH2LLQ?t=323`).

4.1.12 Testing Video

I have uploaded an unlisted video to `youtube.com` going through each test here: `https://www.youtube.com/watch?v=RsrDGrH2LLQ`.

4.2 Alpha-beta pruning test

4.2.1 Purpose

Alpha-beta pruning should give exactly the same minimax given the same input. This test evaluates an Othello position using minimax *without* alpha-beta pruning, minimax *with* alpha-beta pruning, and asserts the equality of the two evaluations.

4.2.2 Objectives affected

This affects objective six. If this test fails, there must be a bug in either my minimax implementation or my implementation of alpha-beta pruning.

4.2.3 Input

I wrote this test using Rust's unit-testing framework. This is the code for the test:

```
#[test]
fn alpha_beta_works() {
     use crate::evaluation::{better_eval, minimax,
    minimax_without_alpha_beta_pruning};
     let board = Board::default();
     let without =
         minimax_without_alpha_beta_pruning(better_eval, &(board, board.each_move())
    , 8);
     let with = minimax(
        better_eval,
        &(board, board.each_move()),
        8,
        i16::MIN, // alpha = -infinity
        i16::MAX, // beta = +infinity
    );
    assert_eq!(without, with);
}
```

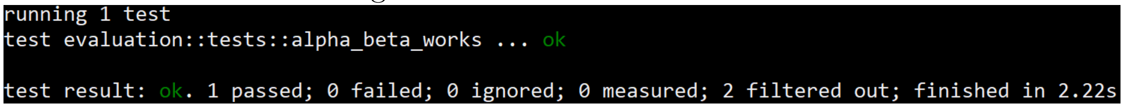
When I enter `cargo test alpha_beta_works` into the command line, this test will be run.

4.2.4 Expected Outcome

I expect the variables `without` and `with` to be equal, which will make the test pass.

4.2.5 Outcome

The test passes. The evidence is in figure 4.4.



```
running 1 test
test evaluation::tests::alpha_beta_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 2 filtered out; finished in 2.22s
```

Figure 4.4: This is the result of entering `cargo test alpha_beta_works` into the command line. The test passes.

Chapter 5

Evaluation

5.1 Fulfilment of Objectives

I have demonstrated that every objective is met in my video (`https://youtu.be/RsrDGrH2LLQ`) and in the testing chapter, so my solution fits the end user's requirements very well.

5.2 End User Feedback

I asked my end user to use the software and fill in a form. The end user's feedback from the form is shown below.

Do you feel that the solution meets the original requirements?

Yes. Everything I mentioned in the interview has been implemented.

Is there anything which you would prefer to have been done differently?

Moves are made and discs are flipped instantly when I click on the board and when the computer makes a move. It would be helpful if there was an optional animation which would show the discs being flipped to make it more obvious how the state of the board changes from move to move.

How could the current solution be improved?

The GUI could look nicer, but it works fine. Apart form that, there is nothing I can think of which apart from big additional features, such as multiplayer.

Is the computer player strong enough?

Yes. I have never beaten the computer at a depth of 7, and each move takes less than a second, so I am always adequately entertained.

I think this feedback is quite fair. If I had had more time, I would like to have implemented the things mentioned on the form.

5.3 What Could be Done Differently

If I were to embark on a similar project again I would like to spend more time optimizing the evaluation function and experimenting with different search algorithms. I would also like to implement some animations in the graphical user interface which would make it obvious when a move has been made by either player; at the moment moves are made instantly and this can be slightly confusing. I have never written software which uses the internet, so implementing a multiplayer system which the end user mentioned on the form would be new to me, and possibly quite difficult and time consuming, but I think it would be a good thing to learn (if I had the time).