Reversi Web Application

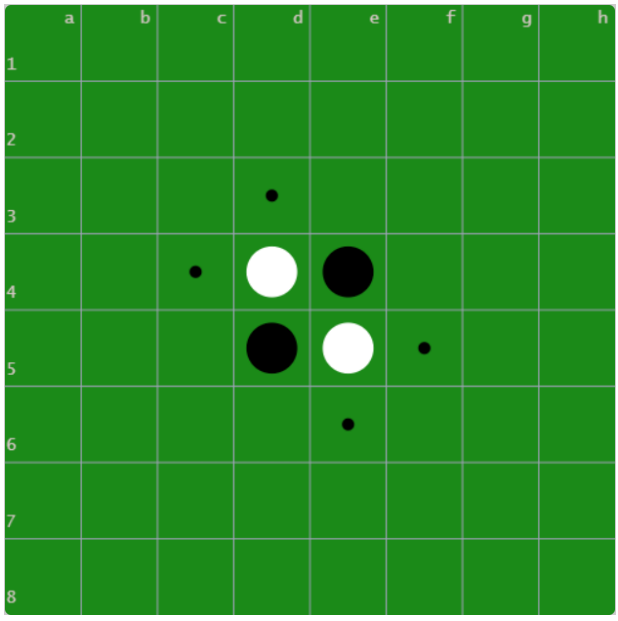
Table of Contents

Analysis

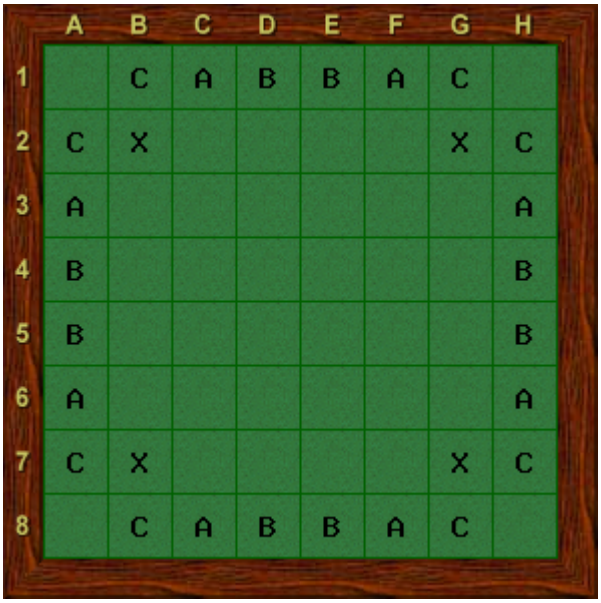
In this project, I aim to create a program to play the game of Reversi, a two-player strategy game played on an 8x8 board. Reversi is a perfect information, turn based game in which players aim to flip the opponent's discs by enclosing them with their own coloured discs. Since Reversi is not as popular as board games like chess, it is often difficult for Reversi players to find opponents of a similar skill level and, due to the entirely abstract nature of the game, it can be very difficult for beginner players to improve their strategy. By creating a web application in which users can play against computer players and get a post-game analysis of their play, I aim to solve these two problems faced by beginner to intermediate Reversi players.

Background to the Problem

In order to explore the problem and the nature of the solution, it is first necessary to explain the rules of Reversi and to briefly explore some of the basic strategy. As mentioned above, Reversi is played between two players (referred to as light and dark) on an 8x8 board. The initial position is setup as shown below and dark, whose options are shown by the smaller black dots, is first to move.



Before continuing, it is first useful to introduce some notation. Each square is indexed by a letter and a number where the letters increase rightwards, and the numbers increase downwards. For example, the upper left square is labelled a1 and the lower right square is labelled h8. In addition, due to the symmetry of the board, it is conventional to assign labels to squares which are often strategically equivalent.



As shown in the above diagram, the squares diagonally adjacent to the corners are referred to as X squares and the squares horizontally or vertically adjacent to the corner squares are labelled C squares. Moving towards the middle of the sides of the board, the squares on the opposing side of the C squares to the corners are referred to as A squares. Finally, the pair of squares in the middle of each side of the board are referred to as B squares.

The primary mechanic of Reversi is flipping the opponent's discs to the player's colour by surrounding them between two discs of the player's colour. A move consists of placing a disc of your colour into a square and, in order for a move to be legal, at least one of the opponent's discs must be flipped as a result. Discs may be flipped in more than one direction and all of the enemy discs sandwiched between the player's discs are flipped, so long as they form a continuous line. As long as a player has legal moves, they are obligated to make a move and, if they have no available moves, they are forced to pass their turn. Once every square is occupied or neither player has a legal move, the game ends and the player with the most discs of their colour on the board is declared the winner.

Having established the rules and win conditions of the game, let us briefly explore some of the basic strategy. It is necessary to have this basic understanding both in order to assess how well the program is playing and also to inform the design of the heuristic evaluation function (this will be discussed in depth later in this report). Since the end objective of the game is to have the most discs, it is reasonable to think that a good move is a move that flips the most discs. However, it turns out that, apart from in the end game, the opposite is generally true. Although players cannot pass their turn if they have legal moves available, it is often preferable to play a move that is essentially equivalent to passing and affects the board as little as possible. In Reversi strategy, this is known as a quiet move and, conversely, a move that flips a lot of discs and has a large impact on the board is considered to be loud. The distinction between quiet and loud moves is more subtle than this at higher levels of play but, for the purposes of this project, this distinction is not significant, and I am not knowledgeable enough about Reversi strategy to discuss it here. In addition to playing quiet moves, another core strategy is to gain stable discs, a stable disc being one which can never be

flipped. The corner squares are always stable and, as such, taking corners is an important aspect of the game, but other squares may also become stable depending on the state of the board.

Problems Faced by End User

Having discussed the fundamentals of Reversi, there remains the question of who the user of this application will be and what functionality is therefore required of it. As for the nature of the user, this application will be targeted towards beginner to intermediate Reversi players who perhaps lack opponents of a similar skill level or who want to improve their Reversi playing skills. This application is unlikely to be especially useful for advanced players as they will demand a skill level in an opponent that will likely be infeasible to achieve in this project. Although I consider myself to be part of the target user base, in order to gain another perspective as to what problems are faced by the user base, I will ask my mother, who is a casual Reversi player, what problems and annoyances she faces in playing Reversi and what features would therefore be desirable.

Following my conversation with this potential user, the following problems were identified:

- Unless one knows a lot of people who play Reversi, it is often difficult to find opponents of a similar skill level to gain experience against.
- Having lost a game, it is not easy for a beginner to accurately identify their blunders and inaccuracies. In addition, it is difficult to retrospectively identify which moves should have been played instead.
- It is difficult for beginners and intermediate players to identify which areas of gameplay (e.g. openings, middlegame, endgame) are weakest for them.

High Level Objectives for the Solution

Informed by my discussion with the potential user, I will propose the following primary objectives for the program, although throughout the development process, other secondary functionality may be added. Although most of these objectives directly correspond to the problems identified through discussion with the potential user, some are drawn from the functionality of similar currently available applications.

H1. The user should be able to select the difficulty of the computer opponent.
H2. The program should have a post-game analysis feature which shows how well the player played throughout the game, allowing the user to identify which phases of the game should be improved.
H3. The application should be able to identify blunders and suggest better moves. This functionality should be available in the post-game analysis feature.
H4. The program should be able to defeat a typical user on its maximum difficulty setting. In order to make this requirement measurable, I will propose that it should be able to defeat my mother, who is representative of the typical user.
H5. The program should support the following match setups and allow the user to decide who plays dark in each instance:

    a. Human vs Human
    b. Human vs Computer
    c. Computer vs Computer

H6. The user should be able to interact with the program via a user-friendly graphical user interface.

Low Level Requirements

For the purposes of testing, I will also propose the following low level, measurable requirements for the solution.

L1. Upon the webpage loading, the user should be presented with an empty board and an option to start a new game.
L2. Upon clicking a 'new game' button, the user should be able to select whether each player will be a human or computer player, as well as selecting the depth to which computer players will search.
L3. Upon clicking 'start game', the game setup window should close, and the game should begin.
L4. Given a board position, the application should correctly determine a list of legal moves for a given player.
L5. At the start of each turn, the legal moves for the active player should be clearly displayed.
L6. If it is the turn of a human player, the user should be able to place a disc by clicking on a square.
    a. If the square clicked corresponds to a legal move, the correct colour disc should be drawn onto that square on the board.
    b. If the square clicked does not correspond to a legal move, the application should do nothing.
L7. If it is the turn of a computer player, the application should calculate a move using an alpha-beta search to the depth specified by the user or by using a book of predefined openings.
    a. The program should check whether the game is 'in book' and, if so, randomly choose a move which is consistent with an opening in the opening book.
    b. The program should be able to assign a numerical desirability score to a board position using a heuristic evaluation function.
    c. Using the values assigned to leaf nodes, the program should correctly perform an alpha-beta search to determine the best move.
L8. If no legal moves are available to the active player, the program should notify the user of this and toggle the active player to be the other player.
L9. After a move is inputted, the relevant discs should be flipped, and the board should be redrawn to reflect this.
L10. After each move, the application should update a game history table to display what moves have been played.
L11. After each move, the name of the longest opening in the opening book which is consistent with the game should be displayed.
L12. After each move, the square in which a move was most recently played should be highlighted by being made slightly darker.

L13. If neither player has a legal move (i.e. if the game has ended), a post-game window should pop up displaying the result
L14. The post-game popup window should also display an analysis table for the game. For each move, the analysis table should display:
    a. The desirability of the resulting position from the perspective of the player making the move, according to an alpha-beta search at a search depth of three[1].
    b. A qualitative description of the move (i.e. book move, mistake, good move, correct move)
    c. The move that should have been played according to an alpha-beta search at a search depth of three[1].

Research of Existing Reversi Programs and Potential Solutions

In order to inform the design of the solution to the problem, I will first survey the approaches of existing programs and research their fundamental principles of operation. Since the creation of a reasonably strong Reversi engine (H4) is the most important objective to the overall functionality of the application, and indeed the most difficult to satisfy, this will be the focus of the research. Due to the abundance of information on the Logistello program, most of my research relates to Logistello, although most strong Reversi engines will operate on similar principles. Indeed, in addition to Logistello, I also researched the WZebra and BILL programs.

Game Trees

The fundamental principle on which all strong Reversi engines operate is that of exploring the game tree down to a specified depth and then, through a process of backwards induction, deciding which move from the current position will lead to the most favourable position later in the game. Here, a game tree refers to a tree in which each node represents a possible game state (board position in the context of Reversi) and an edge from node A to node B represents a legal move from position A which results in position B. For non-leaf nodes, the value of a node is a function of its children but, for leaf nodes, a value must be directly computed which reflects how favourable that position is.

In order for a game to be solved through this process of backwards induction, where 'solved' means that the outcome is known under perfect play, the program must search to the end of the game. That is to say, the leaf nodes of the game tree will be ended game states. In this case, it is trivial to assign a value to the leaf nodes; one scheme could be +1 for a win, 0 for a draw and -1 for a loss. For simple games such as tic tac toe, which have few moves in a game and few options from each position, it is computationally feasible to solve the game in this way. However, for more complex games such as Reversi, this is not computationally feasible and, as such, it is necessary to limit the search depth to look a certain number of moves, or 'plies' ahead. Of course, since the leaf nodes of the resulting game tree will no longer be ended games, it is no longer trivial to assign each leaf game

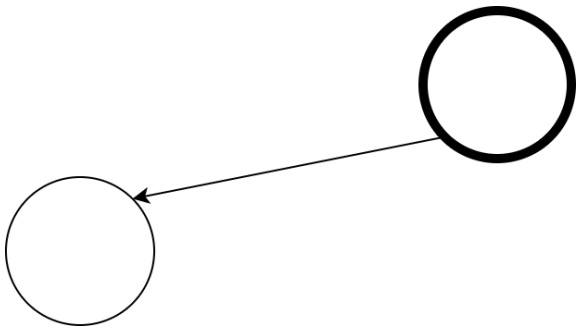
[1] A search depth of three was chosen as it results in reasonably accurate analysis while maintaining fairly short load times.

state a value representing how desirable it is. In order to solve this problem, strong Reversi engines such as Logistello, BILL, and WZebra use a heuristic evaluation function which can be applied to a leaf game state to determine how desirable that position is for the computer player. As the name suggests, these evaluation functions use existing knowledge about Reversi strategy to assess the desirability of a position. The evaluation functions used by the strongest Reversi engines take into account a very large number of metrics relating to a position but the most fundamental metrics that are considered are mobility (the number of moves available), the number of discs adjacent to empty squares, and whether the corners have been occupied. As the evaluation function is arguably the most important aspect of a Reversi engine, I will discuss it more depth later. First however, I will explore the specific functionality of the game tree search algorithm.

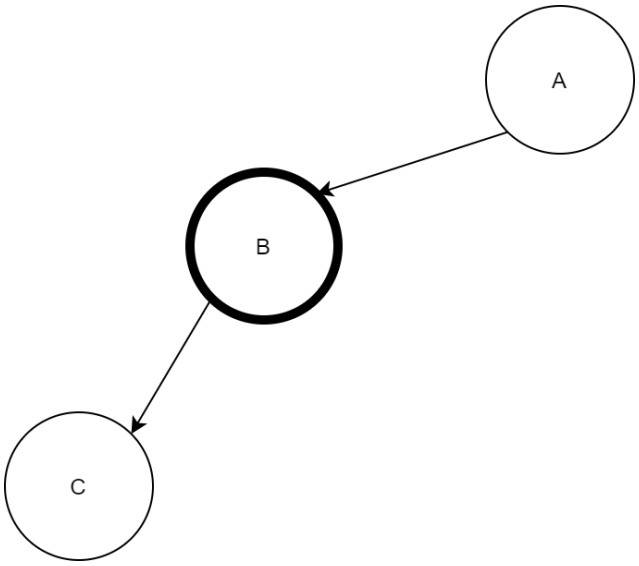
Minimax

The commonest algorithm used to traverse the game tree and decide upon the best move from the root game state is the minimax algorithm. The algorithm recursively applies itself to the child nodes of the root node on which it is called and eventually returns a value which represents the minimum value of the opponent's maximum payoff. Since Reversi is a zero-sum game in the sense that a better position for player A is equivalent to a worse position for player B, the minimax value is equivalent to the maximum value of the player's minimum payoff. In other words, the minimax algorithm seeks to find the maximum payoff that the player is assured of even if the opponent plays perfectly. Since the need to use precise language results in a worded explanation of the algorithm being rather difficult to conceptualise, I will clarify the operation of the algorithm through a series of diagrams.

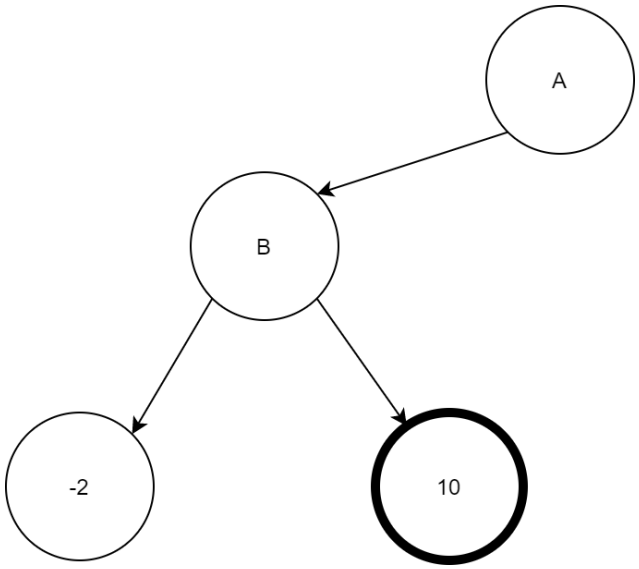
Let us consider for the sake of simplicity a game in which there are always two legal moves from each position. When the minimax algorithm is first applied to a particular game state it will apply itself to each of the game states that are immediately reachable from that game state in sequence. The diagram illustrates this start of the process in which algorithm considers the first child node of the root node.



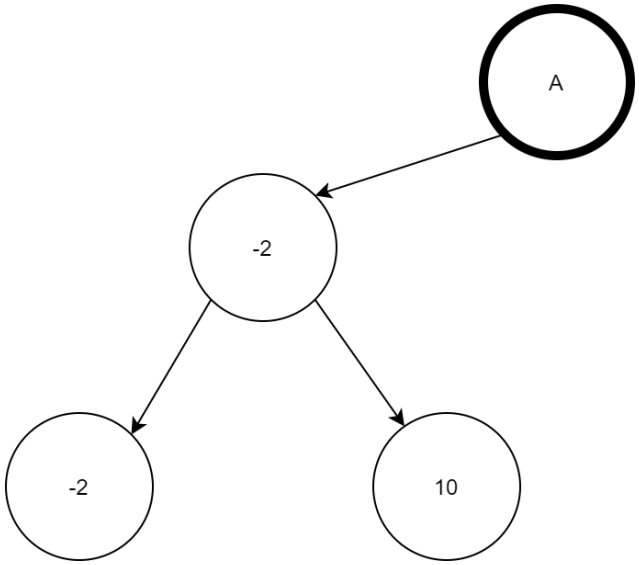
In this diagram, the node with the thicker outline is the node currently being considered. Since the minimax algorithm dictates that the value of a node is some function of the values of its child nodes, which are as yet unknown, the algorithm now applies itself to the first child node.



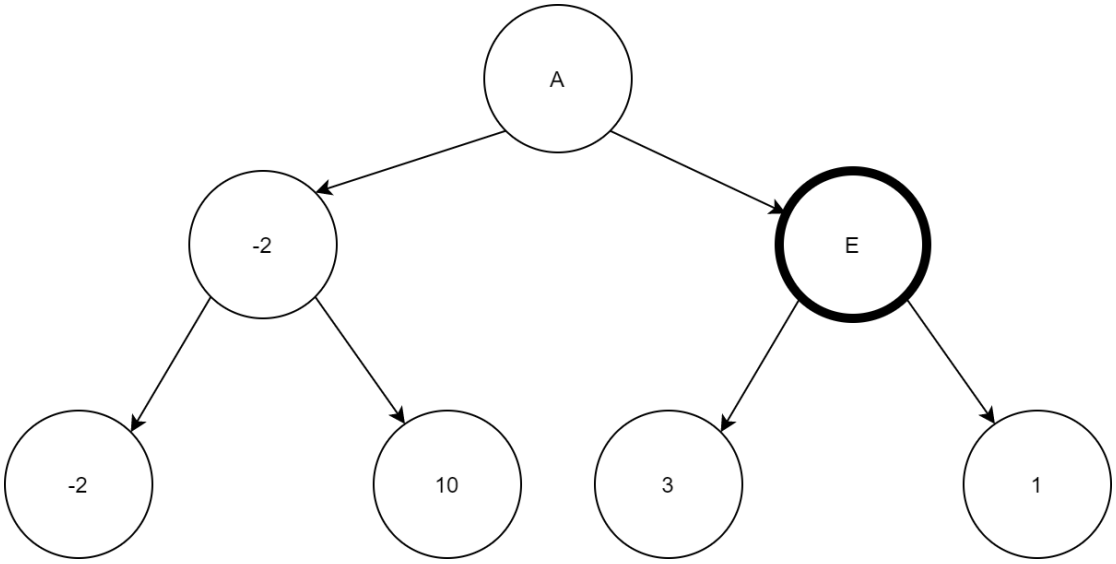
Since our current depth will be enough to demonstrate the concept, let us suppose that this minimax call has been limited to a search depth of 2 which, in the context of the game, corresponds to looking 2 moves or 'plies' ahead. Since the algorithm has searched deep enough, when node C becomes the 'active' node being considered, the recursion will have reached its base case. Now, instead of seeking to calculate the value of the node as a function of the values of its children, the algorithm directly calculates the value of node C using the heuristic evaluation function. For the sake of argument, let us suppose that C is assigned a value of -2, meaning that this position slightly favours the opponent. Having calculated the value of a leaf node, the algorithm will return to considering its parent node (B in this case), from which it will resume the exploration of $B's$ children. Let us suppose that C's sibling node is assigned a value of 10 by the evaluation function.



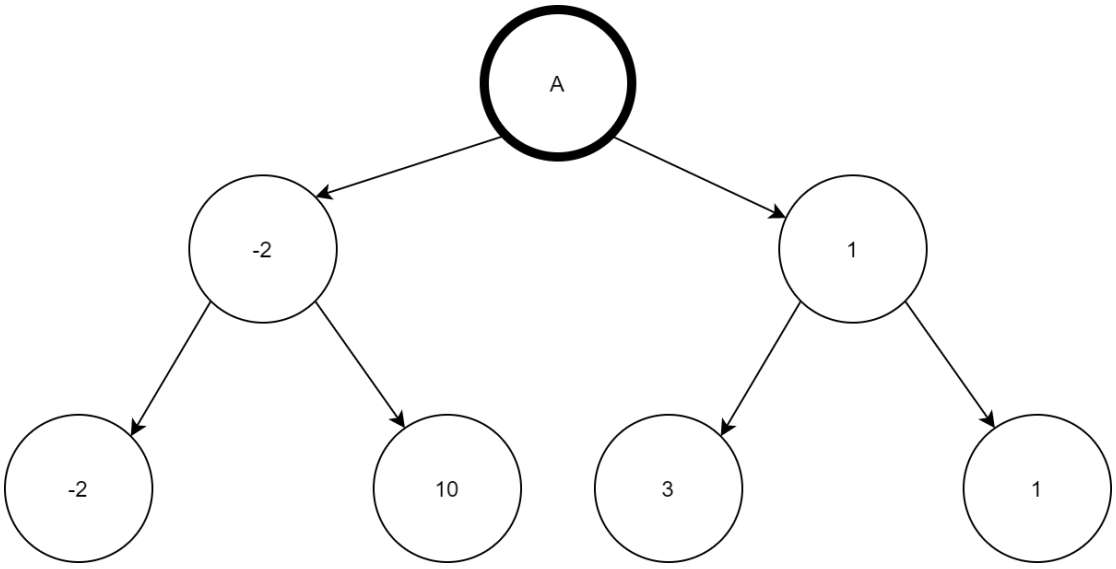
Note that, in the diagram, the alphabetic labels are replaced by the value of the node when it is discovered but the labels are not meant to act as variables representing unknown values. Once again, having reached its base case, the algorithm moves up a level and B becomes the 'active' node. This time however, since B has no more unexplored children, it can be assigned a value based on the values of its children. Since edges linking B to its children represent potential moves for the opponent and the opponent will choose the move with the lowest value, the value of B is given by the minimum of its children's values (-2 in this example). Having assigned a value to B the algorithm will once again move up a level and resume considering node A.



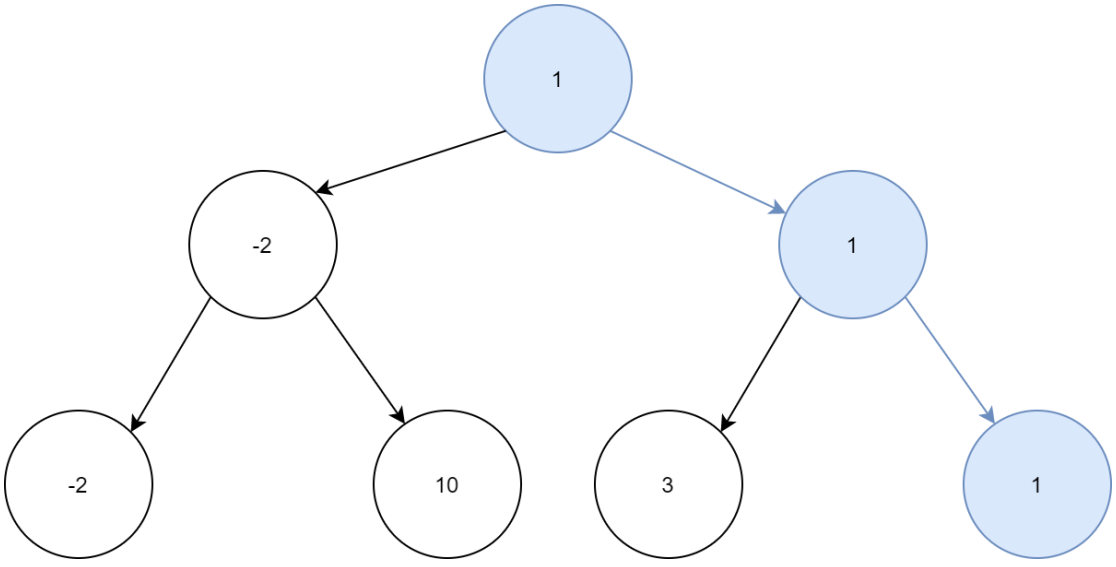
Next, A's right subtree is explored in the same depth first way and so I will only show the key stages in this process.



Here, since it is the opponent's turn in the position represented by E, the value of E is taken to be the minimum of its children. For this reason, the opponent is referred to as the minimising player. In general, the value of a node with an even level is given by the minimum of its children and the value of a node with an odd level is given by the maximum of its children's values. The level of a node is defined to be $1 +$ the number of edges between the node in question and the root node.



Since A is on level 1, its value is given by the maximum of its children's values. This reflects the fact that, at A, it is the computer's move and so it will choose the move which promises the largest payoff. As such, the computer is the maximising player.



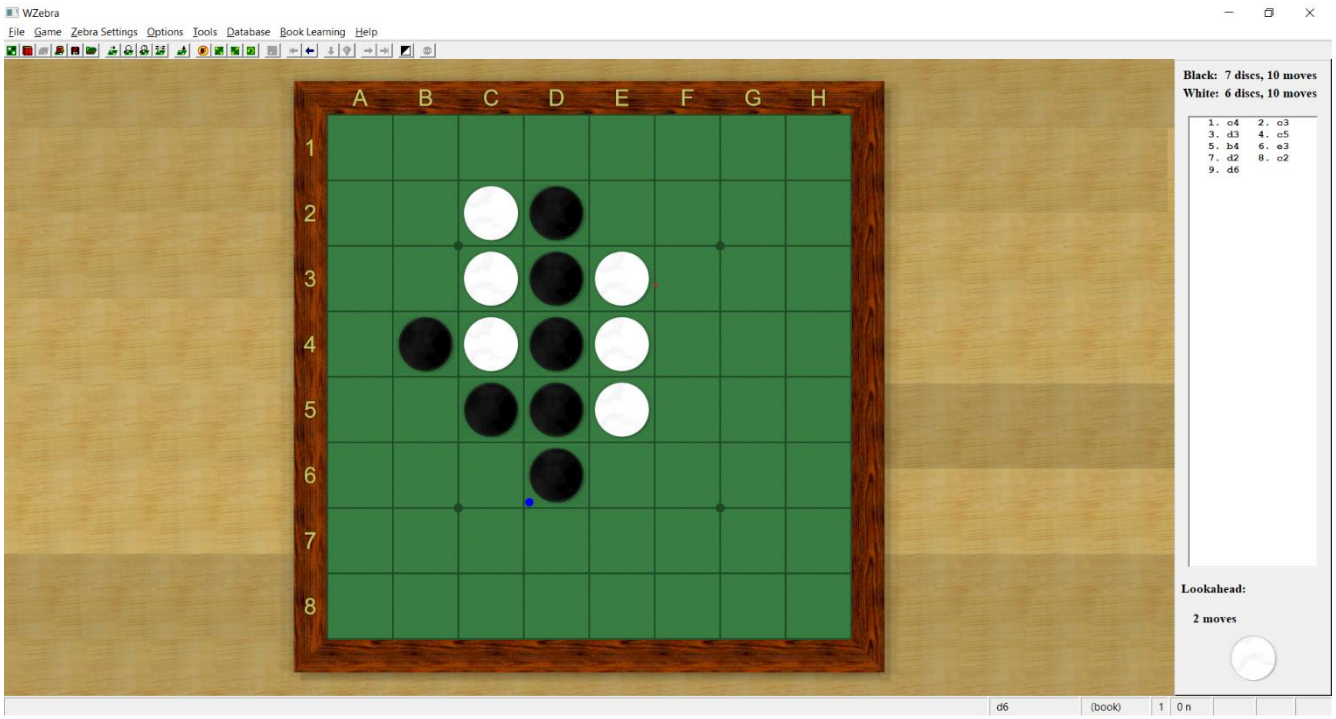
The completed game tree reveals that the value of the root node, which represents the position to which the minimax algorithm was applied, is 1. That is to say, the position to which the minimax algorithm was applied marginally favours the computer. Furthermore, we know that the optimal sequence of moves for both players is the line of play highlighted in blue. It is interesting to note that 1 is not the maximum of all the leaf nodes' values and there are several possible leaf positions that are more favourable for the computer. However, they will never be reached in perfect play as, if

the computer plays the variation represented by A's left subtree hoping to reach the position valued at +10, this will be punished by a good opponent and the leaf position reached will be valued at -2.

User Interface and Experience

In addition to researching the process by which the program actually plays Reversi, I will also survey the user interfaces of various existing Reversi applications, as well as some equivalent applications for other board games. This will allow me identify weaknesses in the user experience of existing applications as well as give me inspiration as to possible designs for the web application.

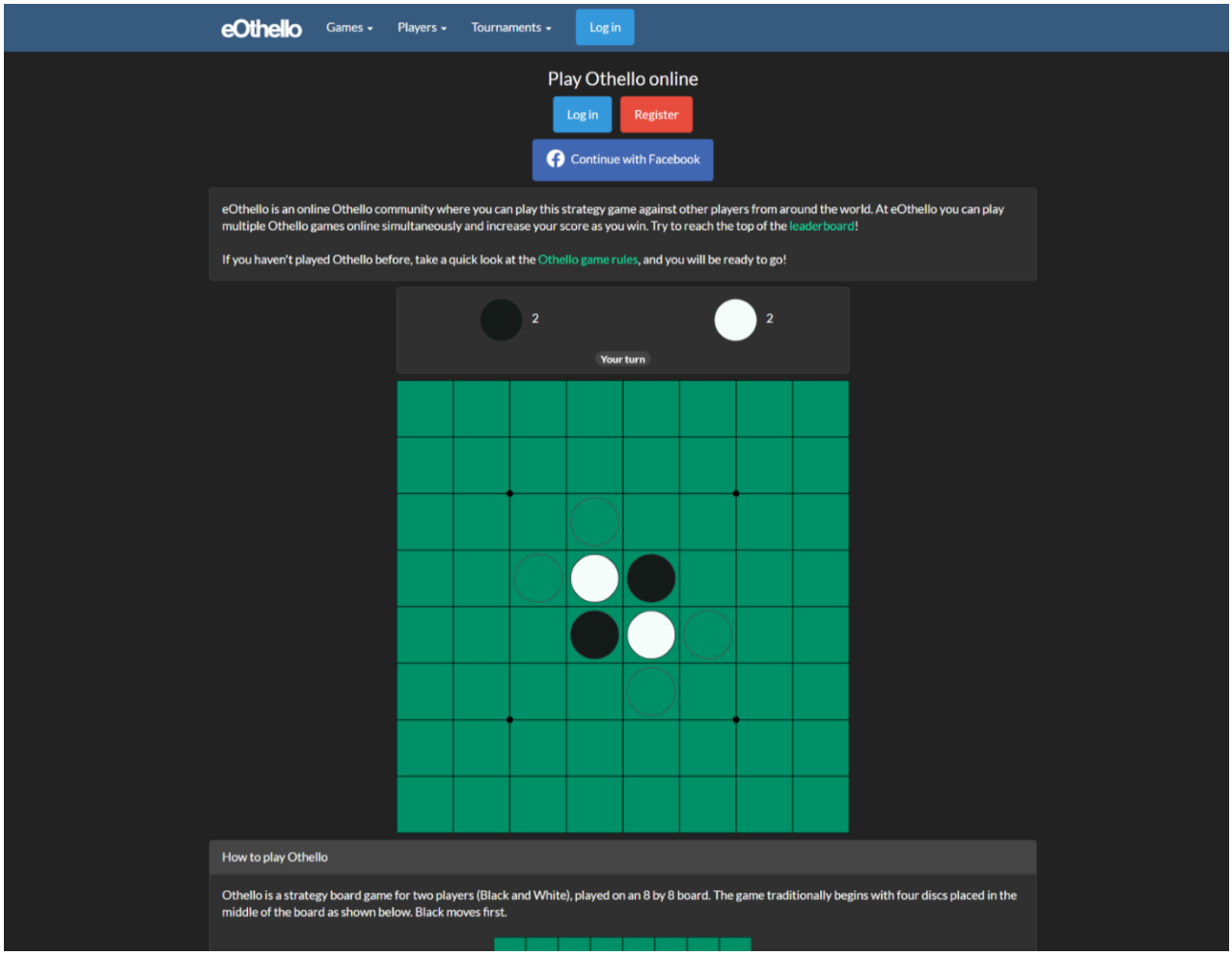
WZebra



WZebra is a desktop application which features one of the strongest freely available Reversi engines. In terms of the board, the UI is intuitive and uncomplicated, if slightly outdated in its use of wood textures and shadows. It helpfully displays a blue dot in the lower left corner of the square in which a disc was last place as well as displaying the game transcript in Reversi notation on a panel on the right.

One potential weakness of WZebra's UI is that, although the core features are intuitive to use, the more advanced features such as controlling the program's use of its opening book and changing the search depth are somewhat hidden away in a cluttered menu system.

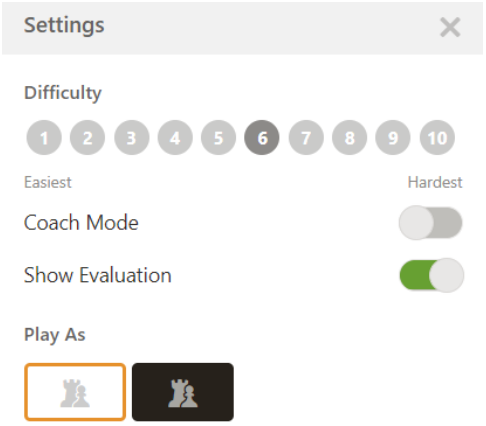
eOthello



Overall eOthello has a much more modern design than WZebra, which appears to have been achieved through the use of Bootstrap or a similar framework. However, it does suffer from cluttering the page with various login buttons and messages to the extent that the page has to be scrolled down in order to see the full Reversi board. Similar to WZebra, the most recently played disc is marked with a small red dot and the inner 5x5 grid is highlighted with black dots at the corner. An additional feature of the eOthello website is that it shows the legal moves available to the player whose move it currently is, which can be useful for beginners are new to the game. Further down the page, the website also explains the rules of Othello, which, although potentially useful, could benefit from being on a different page rather than further down the same page as the board.

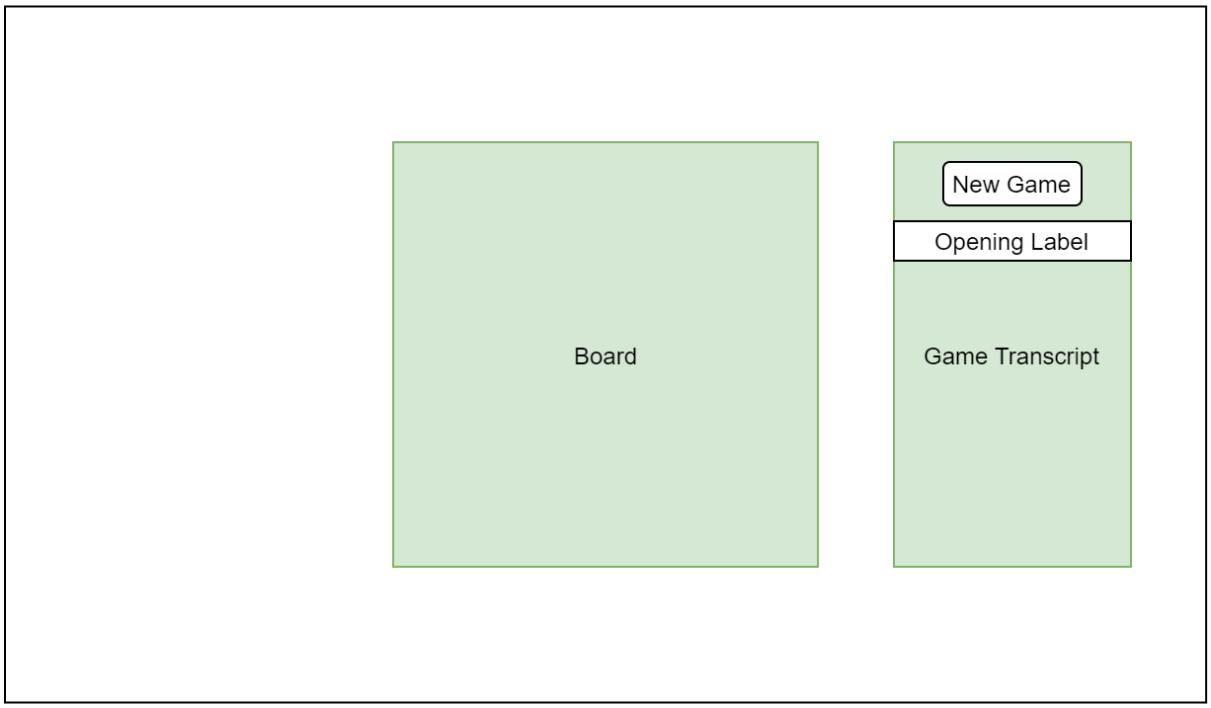
Chess.com

Although chess.com is a chess website rather than a Reversi website, its design and user interface is, in my opinion, excellent and is therefore useful as a source of inspiration for the design of the solution. The design manages to be modern and minimalistic whilst providing plenty of useful information. For example, the website uses the chess engine's opening book to determine what opening and, more specifically which variation is being played. This is displayed in the information panel on the right of the screen (the text in the screenshot reads *Indian Game: Yusupov-Rubinstein System).* As in WZebra, a transcript of the game is provided and the most recent move is shown by highlighting the relevant squares on the board. Finally, the website's game setup menu depicted below is clean and uncomplicated while still providing enough scope for customisation.

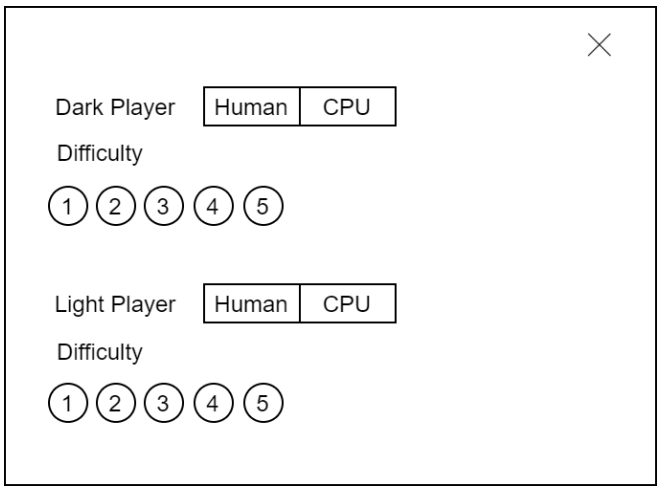


Wireframe Mock-up

Informed by surveying the user interface of existing solutions, I have created a crude wireframe mock-up of the planned design of the webpage.

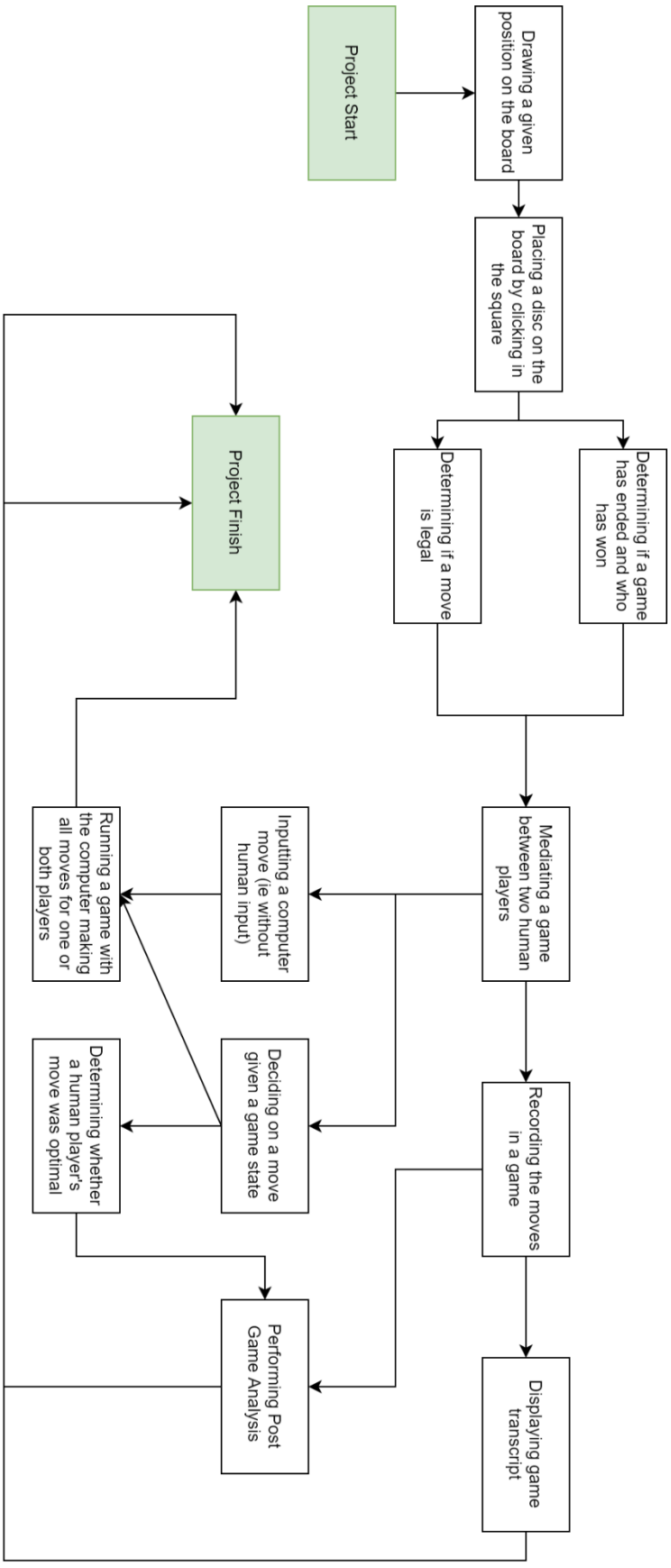


I plan for the game setup menu to be a popup menu laid out as roughly illustrated below.



The number of degrees of difficulty depicted here is arbitrary as I do not yet know how many there will be in the final solution.

Breakdown of Work and Project Network



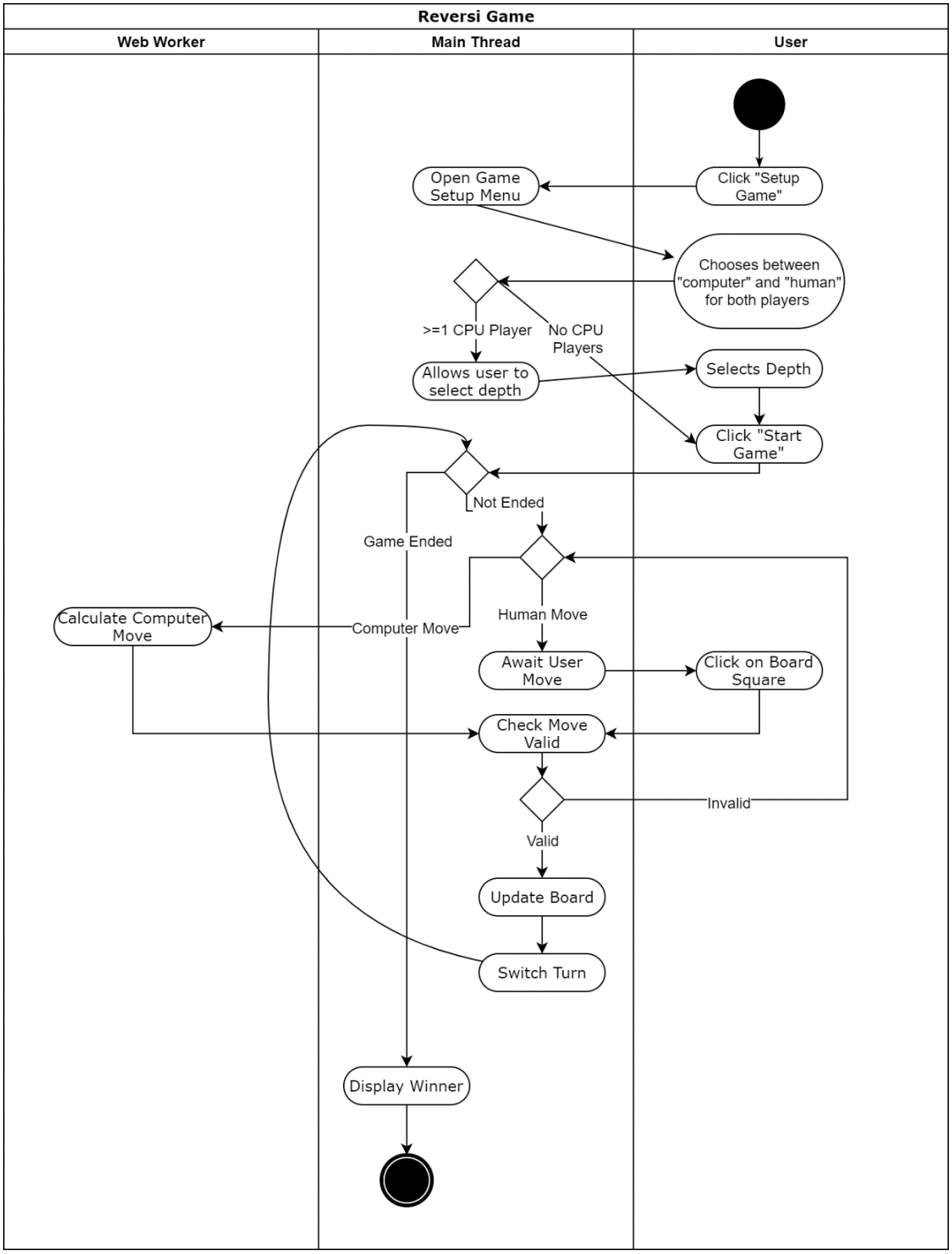
In this project network diagram, an arrow from task A to task B implies that starting task B is contingent on the completion of task A.

Design

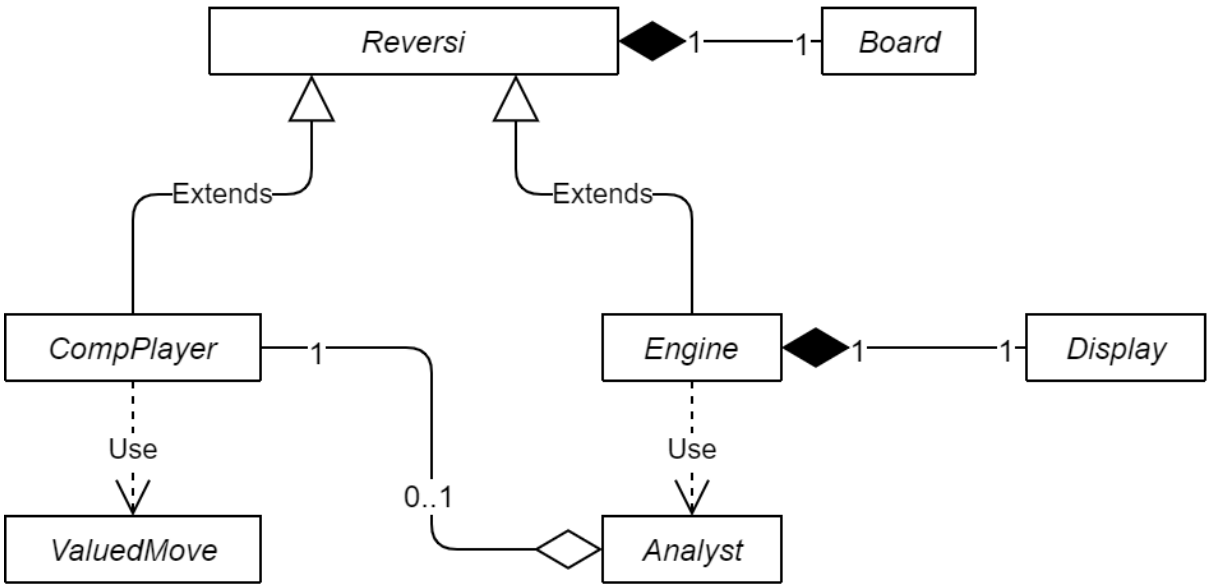
High Level System Design

User/System Swimlane Diagram

The swim lane diagram below shows the logical flow of the various processes involved in running a game of Reversi. The lanes represent separation of responsibility and so all processes that are performed by the user are in the lane labelled ‘User’. The program as a whole is modelled as two separate actors in this diagram, with the two lanes representing two concurrent threads. As evident in the diagram, the majority of processes are performed by the main thread but, in order to ensure that the user interface does not freeze while the program is ‘thinking’, the computation of the best move using the minimax algorithm is performed asynchronously by a web worker.


Reversi Game
Web Worker
Main Thread
User
Click "Setup Game"
Open Game Setup Menu
Chooses between "computer" and "human" for both players
>=1 CPU Player
No CPU Players
Allows user to select depth
Selects Depth
Click "Start Game"
Not Ended
Game Ended
Human Move
Computer Move
Calculate Computer Move
Await User Move
Click on Board Square
Check Move Valid
Invalid
Valid
Update Board
Switch Turn
Display Winner

Class Diagram



The above diagram illustrates the relationships between the various classes that will be used in the project, as well as the nature of these relationships. Since some of the classes have many attributes and methods, only the class names have been shown in order to ensure that the diagram remains readable. I shall now describe in more detail the function of these classes, as well as list each class' attributes and methods.

The Reversi Class

Reversi
+ board: object + gameRunning: boolean + darkTurn: boolean + missedTurns: number + freeTiles: number + gameHistory: string + inBook: boolean + openings: array
+ turnToken(): number + gameEnded(array): boolean + insertToken(array, array): boolean + findLegalMoves(array, number): array + findNeighbours(array): array + findOccupiedCells(array): array + generateSurroundVectors(): array + validateNeighbours(array, array, number): array + nextCellInRun(array, array): array + deleteDuplicates(array): array + onBoard(array): boolean + findRuns(array, array, number): array + flipTokens(array, array, number): undefined + countTokens(object): [number, number] + handlePass(): undefined + getRandomMove(): array + loadOpenings(function): undefined + callback(object, object): undefined + checkInBook(): array + getBookMove(array): array + findPossibleOpenings(string): array + filterLongerOpenings(array): array + getCompletedOpening(array): array + machineForm(array): array + humanForm(array): + mockPlay(array, array, number): array + tokenAt(array, array): number

The Reversi class represents a game of Reversi at a fairly abstract level and, as such, contains attributes which pertain to the state of the game such as whose turn it currently is, whether the game thus far is consistent with a known opening (i.e. 'in book'), and whether the game is still running. Its methods are the various processes that must be performed in order to progress the game such as inserting discs and flipping tokens. The Reversi class is never actually instantiated, but rather serves to be the parent of the Engine and CompPlayer classes, both of which are instantiated. This structure of inheritance is primarily motivated by the need for many methods to be shared between the Engine and CompPlayer classes which, in this design, is achieved through these methods being inherited from the Reversi class. From a philosophical point of view, this class structure reflects the fact that, in a sense, both the Engine and CompPlayer classes represent different types of Reversi games, with the Engine class representing a concrete game with which the user interacts, and the CompPlayer class representing a game that is simulated as part of the computer player's thought process. This class is composed of an instance of the Board class which is stored as an attribute of the Reversi class. Since a board only exists within the context of a game, this association constitutes composition as opposed to aggregation.

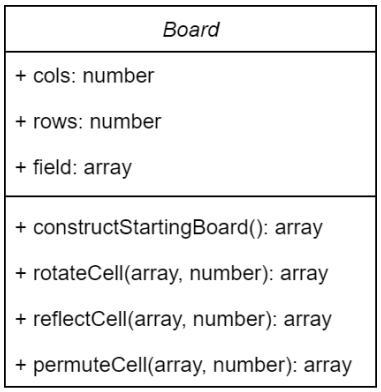
The Engine Class

Engine
+ display: object + darkPlayer: string + lightPlayer: string + darkDepth: number + lightDepth: number + aiDelay: number + useBook: boolean + thinking: boolean
+ initDepthSelect(): undefined + initPlayerSelect(): undefined + updateOpeningLabel(): undefined + updateGameTable(): undefined + initGameTable(): undefined + initAnalysisTable(): undefined + draw(): undefined + handleTurn(): undefined + initiateMinimax(): undefined + createMinimaxWorker(array): undefined + processCompMove(): undefined + executeMove(array): undefined + processHumanMove(): undefined + getSquare(object): array + handleGameEnd(): undefined + analyseGame(number): undefined + populateAnalysisTable(array): undefined

As mentioned above, the Engine class is responsible for the logic of a Reversi game within the context of the application, and the user's interaction with it. The primary distinction between this class and its parent class Reversi, is that the Reversi class exists at a highly abstract level whereas the Engine class concerns itself with the concrete game that is displayed to the user and, as such, handles tasks relating to the interface between the game and the user. As a result, the Engine class contains attributes such as the type (human or computer) of the two players, the search depth used, if relevant, and an instance of the Display class. The methods of the Engine class are mainly to do with the flow of data between the user and the application such as drawing the board, handling a user click and displaying various secondary information such as the opening being played. In addition to being composed of a singular instance of the Display class, the Engine class has a dependency

upon the Analyst class, of which it creates an instance after the game ends in order to provide analysis to the user.

The Board Class



The Board class is relatively self-explanatory in that it represents a board and all of the discs on it. Its methods pertain to various transformations of the board which are useful for checking whether an opening is being played in a different orientation to the version listed in the opening book. This is because a Reversi board, with its initial setup has four symmetries, remaining unchanged after a rotation of 180 degrees and a flip about either diagonal. The constructor method of the class takes a number as an argument which represents the size of the board. For example, a Board(n) instantiates a Board object with an n by n board, where n is a multiple of two. The option to play with different board sizes is not available to the user and so, in the application, n will always equal 8. However, it will sometimes be useful to test the minimax function on a 4x4 or 6x6 board in order to reduce the complexity of the problem.

The CompPlayer Class

CompPlayer
+ evaluateByWeight(array, number): number
+ generateWeightingMask(): array
+ cornerSquare(array, number): Boolean
+ cSquare(array, number): Boolean
+ aOrBSquare(array, number): Boolean
+ xSquare(array, number): Boolean
+ countFrontiers(array): array
+ evaluateByTerritory(array, number): number
+ evaluateByMobility(array, number): number
+ evaluateByFrontiers(array, number): number
+ evaluate(array, number, number): number
+ maxMove(object, object): object
+ minMove(object, object): object
+ alphabeta(array, number, number, number, number, number, boolean): object

The primary responsibility of the CompPlayer class is to calculate the best move from a given position, which it achieves through the use of a minimax tree search with alpha-beta pruning. The CompPlayer class contains no attributes other than the attributes that are inherited from its parent class, Reversi. Its primary method is alphabeta() which actually performs the tree search, whilst the majority of the class' other methods pertain to the computation of the heuristic evaluation function which assigns a value to a position representing how desirable that position would be for the dark player. New CompPlayer objects are instantiated by web workers which asynchronously execute the minimaxWorker.js file. The other occasion on which the CompPlayer class is instantiated is during the creation of a new Analyst object, since objects of the Analyst contain a CompPlayer object as an attribute.

The Analyst Class

Analyst
+ transcript: string + simulator: object + positions: array
+ genPositions(): array + gradePositions(number): object + separateEvaluation(array): [array, array] + constructAnalysisTable(array): array + appraiseMove(number, array, array, string): string + determineOpening(): array + findConsistentOpenings(): array

An instance of the Analyst class essentially serves as an agent which performs the post-game analysis of a Reversi game, abstracting away the complexities of the analysis process from the rest of the program. A single instance of this class is instantiated by the currently in use Engine object at the end of a game.

An Analyst object has three attributes: a transcript of the game to be analysed, a CompPlayer object, and an array of all of the board positions reached in the game. Its methods all perform various tasks involved in analysing the game such as determine what opening was played and assigning a qualitative appraisal to each move. Perhaps the most important method, however, is the gradePositions() methods which applies the heuristic evaluation function to each position that was reached in the game and determines the best move in each position. In order to speed up this process, each position is analysed concurrently using JavaScript's built in Promise objects which are assigned to the return values of web workers.

The Display Class

Display
+ canvas: object + ctx: object + cols: number + rows: number + colWidth: number + rowHeight: number + darkColour: string + lightColour: string
+ drawBoard(object): undefined + drawGrid(): undefined + labelSquares(): undefined + drawDiscs(object): undefined + showLegals(object): undefined + highlightSquare(array): undefined + highlightLastSquare(object): undefined

Each instance of the Engine class contains a Display object as an attribute which is responsible for displaying the game on the HTML canvas, which is itself an attribute of the Display object. As such, the Display class handles drawing the board, displaying the legal moves on a given turn, and highlighting the square on which a disc was most recently played.

The ValuedMove Class

ValuedMove
+ value: number + move: array

The ValuedMove class contains no methods and serves as more of a data type which stores a move along with its associated value as calculated by the tree search algorithm. ValuedMove objects are instantiated exclusively by CompPlayer objects and are used in some of the methods of the CompPlayer class.

Technologies Used

In terms of the technologies used in this project, the Reversi board itself will be rendered using the canvas element in HTML 5 and the game logic and game playing engine will be coded using JavaScript. JavaScript will also be used to manipulate the HTML document and is therefore responsible for realising the dynamic elements of the user interface. In order to more easily select DOM elements, I will also make use of the JQuery library for JavaScript. The aesthetic aspects of the web application and responsiveness to various screen sizes will both be handled by CCS3. The Bootstrap CSS framework will be used in order to speed up the aesthetic aspect of development as this is not intended to be the primary focus of the project. Finally, throughout the development and testing process, the Apache HTTP server will be used.

File Structure

The design for the overall file structure of the project is shown by the following 'tree' command.

```
C:\xampp\xampp2\htdocs\WebReversi>tree /F
Folder PATH listing for volume Windows
Volume serial number is 6A93-F69B
C:.
│   index.html
│
├───books
│       openings.txt
│
├───css
│       main.css
│
└───js
        analyst.js
        board.js
        engine.js
        graphics.js
        main.js
        minimaxWorker.js
        reversi.js
        thinker.js
```

The graphics.js file contains the Display class, while the thinker.js file contains both the CompPlayer and ValuedMove classes. With the exception of main.js and minimaxWorker.js, which are self-explanatory, all other JavaScript files simply contain the classes with which they share a name. The openings.txt file contains a list of the openings in the application's opening book, all of which were taken from http://samsoft.org.uk/reversi/openings.htm. Since the file only contains 77 openings, lookup times will be very short and so it will be satisfactory to store the openings in a simple comma

separated value format. Although storing the openings in a more sophisticated structure such as a hash table would be useful for very large opening books, I do not anticipate this opening book growing in size. As such, the openings will be stored in the following format.

```
C4c3, Diagonal Opening
C4c3D3c5B2, X-square Opening
C4c3D3c5B3, Snake/Peasant
C4c3D3c5B3f3, Lysons
C4c3D3c5B3f4B5b4C6d6F5, Pyramid/Checkerboarding Peasant
C4c3D3c5B4, Heath/Tobidashi
C4c3D3c5B4d2C2f4D6c6F5e6F7, Mimura Variation II
Etc.
```

Here, the capitalisation of the column letter indicates that a move is to be played by dark and, conversely, a move notated with a lowercase letter is to be played by light.

Description of Key Data Structures

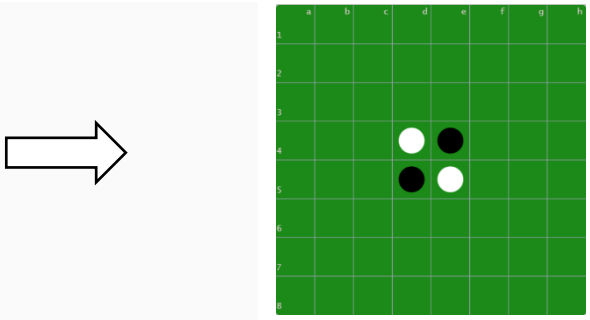
The Game Board

The game board is represented using a two-dimensional array (usually referred to as the 'field' in the code). Each entry in this matrix is either 0, 1, or 2 according to the following scheme.

- 0 ⇒ empty square
- 1 ⇒ dark disc
- 2 ⇒ light disc

As an example, the opening board position is represented as shown below.

```
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 2, 1, 0, 0, 0]
[0, 0, 0, 1, 2, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
```



Description of Key Algorithms

Speeding Up Minimax Using Alpha Beta Pruning

As I discovered during the research section, all of the major currently available Reversi programs use the minimax algorithm to decide on the best move. However, for complex games such as Reversi

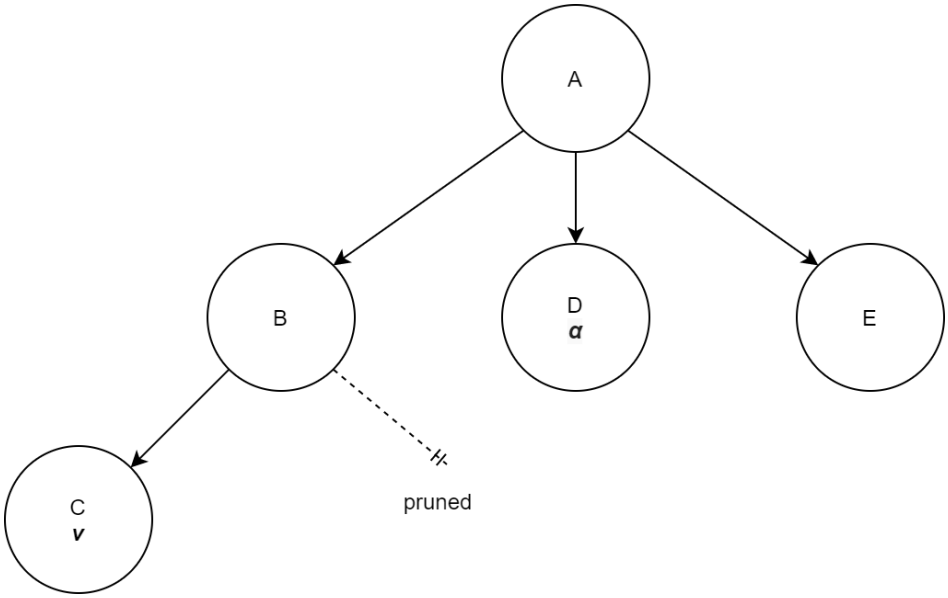
which have a large number of possible moves in each position, the game tree that must be explored grows very quickly as the search depth increases. In fact, Reversi is known to have an average branching factor of 10, meaning that, for each legally reachable position, there are on average 10 possible subsequent positions after one move. Since a tree with a constant branching factor b and maximum depth d has $b + b^2 + b^3 + \cdots + b^d$ non root nodes, the time complexity of a naïve minimax implementation in which the entire tree is searched down to a depth d is $O(b^d)$. For Reversi, this gives a time complexity of $O(10^d)$, which will result in search times growing very quickly with search depth. As such, the alpha-beta pruning algorithm will be used to reduce the number of positions that must be evaluated, allowing the program to search deeper in a given time.

The principle behind alpha beta pruning is not dissimilar to the sort of reasoning that human players use when considering which moves to play. For example, suppose that a human has already found a move A which they know to be good and they are now searching for even better moves. They find such a move B and begin to consider what their opponent might do after B is played. If the human player finds a potential response to move B which results in a worse position than move A would do, it is not necessary to consider the other possible responses to B since B will definitely turn out to be a worse option than option than A. To illustrate this more formally, I have presented the pseudocode for the algorithm.

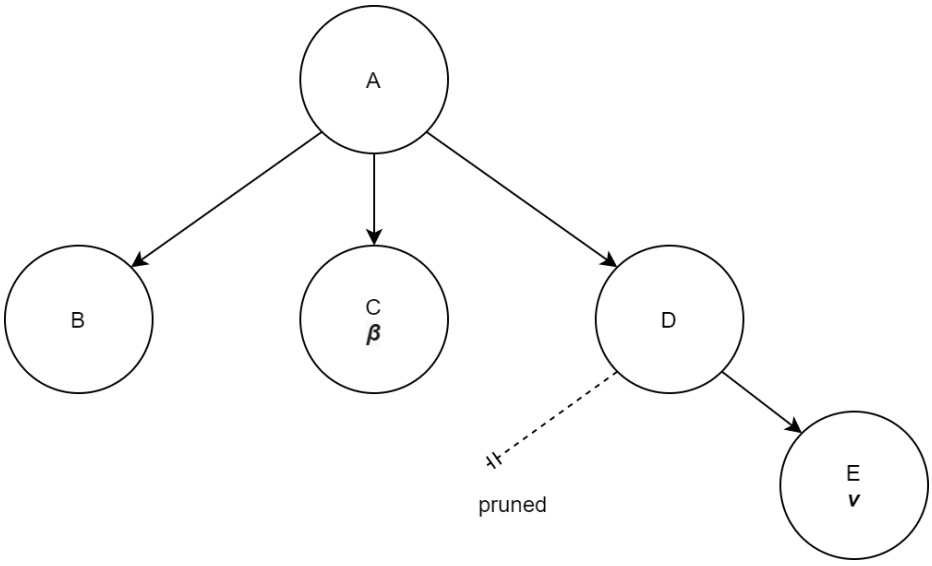
```
function alphabeta(node, depth, alpha, beta, maximisingPlayer)
    if depth is 0 or node is a leaf node
        return heuristic value of node
    if maximisingPlayer
        let value = -infinity
        for each child of node
            value = max(value, alphabeta(child, depth -1, alpha, beta, False))
            alpha = max(value, alpha)
            if alpha <= beta
                break //beta cutoff
        return value
    else //minimising player
        let value = infinity
        for each child of node
            value = min(value, alphabeta(child, depth -1, alpha, beta, True))
            beta = min(value, beta)
            if alpha <= beta
                break //alpha cutoff
        return value

//initial call

alphabeta(root node, depth, -infinity, infinity, True)
```

As shown in the pseudocode, the alpha-beta pruning algorithm keeps tracks of two values, α and β, which represent the minimum payoff that the maximising player is guaranteed of and the maximum payoff that the minimising player is guaranteed of respectively. When searching for the minimum

valued move, if a node C is assigned a value of v such that $v < \alpha$, there is no need to explore the sibling nodes of C as a better option is known to be available to the maximising player. This process, which is referred to as an α cutoff, is illustrated in the following diagram.



Suppose that, as shown in the diagram, nodes E and D have both already been searched and the best move found so far is the move to node D. The maximising player is, therefore, guaranteed a payoff of at least the value of D; we will call this value α. Suppose that, when exploring the children of B, node C is found to have a value v, where $v < \alpha$. Even if the other children of B are more favourable for the maximising player, the minimising player can always move to C from B, resulting in a less favourable position for the maximising player than if they had moved to D from A. If, on the other hand, the children of B turn out to be even less desirable for the maximising player than C, then the value of B will turn out to be even less than v which is, in itself, less than α. Therefore, irrespective of the other children B, no further possibilities from B need to be explored since D is guaranteed to be a better option for the maximising player than B. An equivalent process known as a β cutoff can occur when searching for the maximum valued move and this is illustrated in the following diagram.



Suppose that, at position A, it is the turn of the minimising player and that, having searched nodes B and C, node C is found to give the lowest value so far. Therefore, the minimising player is guaranteed to reach a position with a value of, at most β, where β is the value of node C. If node E is then found to have a value v, where $v > \beta$, then by the same logic as before, the value of node D will turn out to be at least v and so D is guaranteed to be a worse option for the minimising player than C. As such, the other branches of D are pruned as they need not be explored.

It is possible that the alpha-beta pruning algorithm does not prune any branches, in which case its time complexity is $O(b^d)$, the same as the naïve minimax implementation. This worst case arises when moves are considered in the order of worst move to best move. However, if moves are considered in the order of best to worst, the maximum number of cutoffs will occur and the time complexity of the search reduces to $O(\sqrt{b^d})$, allowing the alpha-beta algorithm to search twice as deep in the same time compared to a naïve minimax implementation.

The Evaluation Function

As previously explained, since it is infeasible in most cases to search to the very end of a Reversi game, at a certain depth, it is necessary to directly compute a value for a non-endgame board state which represents how desirable that board state for the maximising player. As such, we need to devise some function which takes a board state as an input and outputs an estimate of how desirable that position is for one of the players. Since Reversi is a zero-sum game, knowing how desirable a position for one player instantly confers how desirable it is for the other player. Recently DeepMind's AlphaZero program, which is designed to play chess, shogi and go, approached this problem by training a neural network to determine how desirable a position through self-play only. However, this is approach is beyond the scope of this project and, since a lot of knowledge already exists about what constitutes a strong position in Reversi, it will be feasible to produce a reasonably accurate evaluation function using a heuristic approach that makes use of this knowledge. In this application, three metrics will be used in combination to estimate the desirability of a position:

- Mobility
- Frontiers
- Square Weightings

Mobility refers to the number of moves available to each player. In this program, dark's mobility is given by the number of legal moves available to dark minus those available to light. It is generally held that a position in which a player has many options and their opponent has few is a strong position for that player and, as such, the more positive the mobility, the more positive the evaluation function.

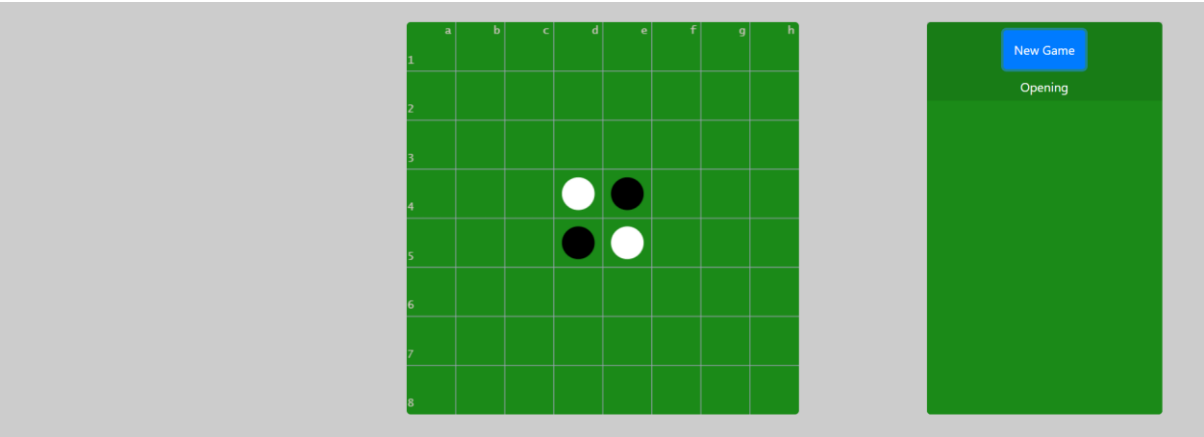
A frontier disc is a disk that borders at least one empty square and conventional Reversi wisdom holds that having a large number of frontier discs is undesirable, especially if these frontier discs are connected to each other to form a 'wall'. Simply put, the reason that frontier discs are to avoided is that they cut off one's access to potentially good moves while increasing the opponent's options.

Incidentally, frontier discs are related to the previously mentioned concept of quiet and loud moves in that a move which creates many new frontier discs is said to be loud and vice versa. As such, when considering how desirable a position is for dark, the evaluation function will calculate $(F_D - F_L)$ where F_D is the number of frontier discs that dark has and F_L is the number of frontier discs that light has. Conversely, when light is the maximising player, the evaluation function will depend on $(F_L - F_D)$.

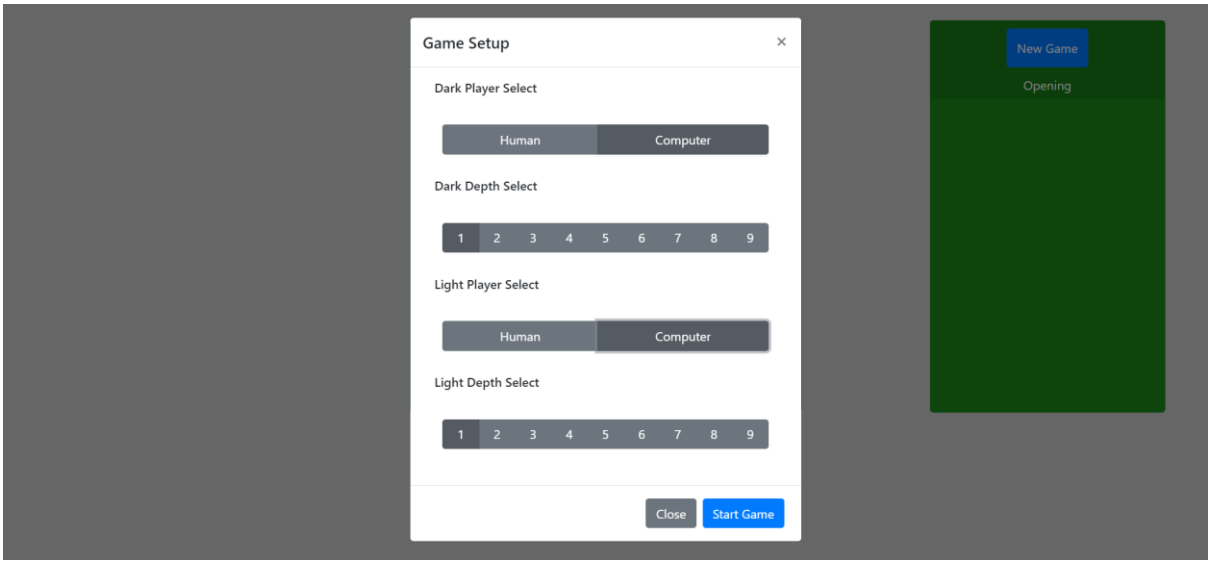
The final factor which will be considered by the evaluation function is that of whether the player occupies certain squares which are known to be generally strong or weak. In other words, each square is assigned a weighting according to how desirable that square is to occupy and a score for dark is calculated by summing the weightings of the squares that are occupied by dark and subtracting the weightings of those that are occupied by light. These weightings are primarily based on the concept of stability, the idea that it is desirable to have discs which can never be flipped. Non corner discs may become stable during the course of a game, but discs on corner squares are always stable regardless of the board position and so they are given a weighting of +100 by the evaluation function. In addition, it is generally the case that playing onto squares adjacent to corners (C or X squares) is to be avoided as it opens up the possibility of the opponent playing onto a corner square. Therefore, C squares and X squares are assigned a weighting of -10. Of course, situations may arise in which playing onto a C or X square is beneficial in that it allows the player to subsequently take a corner square and, since the incentive to take a corner square (+100) is far greater than the disincentive to take X and C squares (-10), the program will correctly evaluate such a play to be beneficial overall. It could be argued that, since the evaluation function already disincentivises the computer player from allowing its opponent to take a corner, it is unnecessary to negatively weight the C and X squares in an effort to avoid this. However, the computer engine faces the horizon problem whereby a seemingly desirable sequence of moves turns out to be undesirable beyond the number of moves that the engine looks ahead. As a result, in the absence of negatively weighted C and X squares, the computer may move to a X or C square, thinking the opponent has no way to subsequently capture a corner square when, in reality, the opponent can capture it in a number of moves that exceeds the search depth of the computer player.

Design of Human Computer Interaction

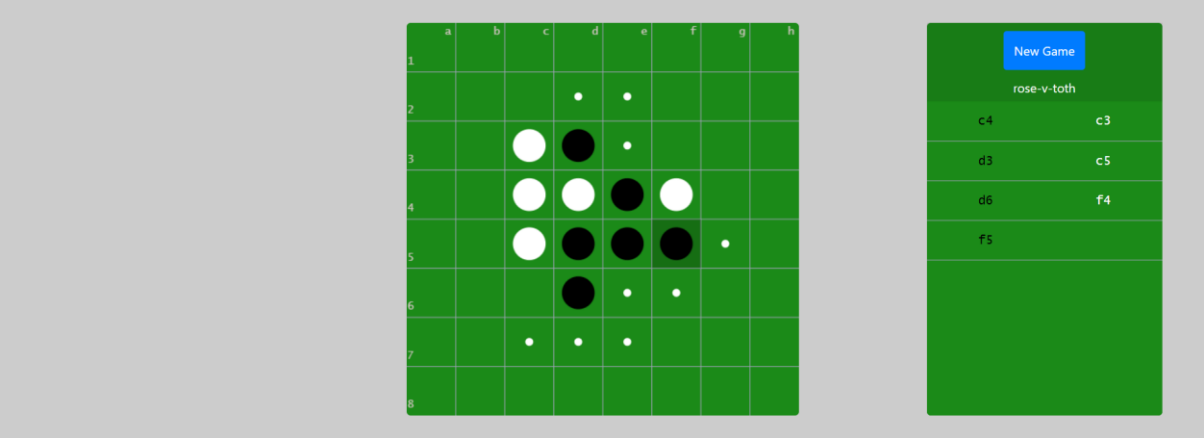
In order to demonstrate the design of the user interface, I will provide screenshots of the final product as this followed the original design.



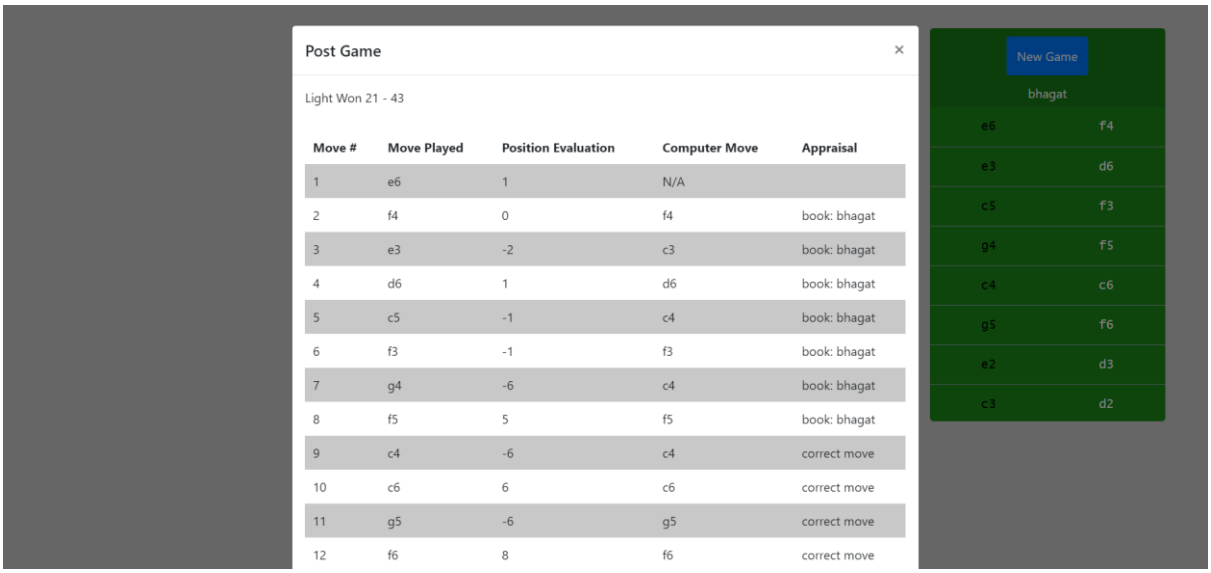
As shown in the above screenshot, the main screen of the application is designed with the board in the centre and a secondary panel on the right. In order to simplify the user experience, the user is presented with only one button – the new game button. When this button is clicked, the user is presented with a game setup popup menu as shown in the screenshot below.



As shown above, the game setup menu allows the user to select the player agent (human/computer) and the search depth via a series of radio buttons. Upon clicking the 'Start Game' button, the game setup menu closes and the game begins. During the game, the user interface appears as shown in the screenshot below.



As shown, during the game, the legal moves for the currently active player are displayed on the board and a transcript of the game is displayed on the right-hand panel. Furthermore, the name of the opening being played is displayed above the game transcript table, also on the right-hand panel. When the game finishes, the user will be presented with the following post-game window which displays the result of the game, as well as a move-by-move analysis of the game.



This post game popup window can be dismissed by clicking outside the window or by clicking a ‘Cancel’ button at the bottom of the table.

Technical Solution

index.html

```
<!doctype html>
<html lang="en">
```

```
<head>
  <meta charset="utf-8">
  <meta name="viewport"
    content="width=device-width, initial-scale=1, shrink-to-fit=no">

  <link rel="stylesheet"
    href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min
.css"
    integrity="sha384-
ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"
    crossorigin="anonymous">
  <link rel="stylesheet" href="css/main.css">
  <script
    src="https://ajax.googleapis.com/ajax/libs/jquery/3.4.1/jquery.min.js">
  </script>
  <title>Reversi</title>
</head>

<body>
  <div class="modal fade" id="setupModal" tabindex="-1" role="dialog"
    aria-labelledby="setupModelTitle" aria-hidden="true">
    <div class="modal-dialog modal-dialog-centelight" role="document">
      <div class="modal-content">
        <div class="modal-header">
          <h5 class="modal-title" id="setupModalTitle">Game Setup
          </h5>
          <button type="button" class="close" data-dismiss="modal"
            aria-label="Close">
            <span aria-hidden="true">&times;</span>
          </button>
        </div>
        <div class="modal-body">
          <div class="container">

            <h6>Dark Player Select</h6>
            <div class="row">
              <div class="col-12">
                <div class="btn-group btn-group-toggle" data-toggle="buttons"
                  id="darkPlayerSelector">
                  <label class="btn btn-secondary active">
                    <input type="radio" name="darkdepth" value="human"
                      id="darkHumanPlayer" checked>
                    Human
                  </label>
                  <label class="btn btn-secondary">
                    <input type="radio" name="darkdepth" value=alphabeta
                      id="darkCompPlayer">
```

```
        Computer
      </label>
    </div>
  </div>
</div>

<h6>Dark Depth Select</h6>
<div class="row">
  <div class="col-12">
    <div class="btn-group btn-group-toggle" data-toggle="buttons"
      id="darkdepthSelector">
      <label class="btn btn-secondary active">
        <input type="radio" name="darkdepth" value=1
          id="darkdepth1" checked>
        1
      </label>
      <label class="btn btn-secondary">
        <input type="radio" name="darkdepth" value=2
          id="darkdepth2"> 2
      </label>
      <label class="btn btn-secondary">
        <input type="radio" name="darkdepth" value=3
          id="darkdepth3"> 3
      </label>
      <label class="btn btn-secondary">
        <input type="radio" name="darkdepth" value=4
          id="darkdepth4"> 4
      </label>
      <label class="btn btn-secondary">
        <input type="radio" name="darkdepth" value=5
          id="darkdepth5"> 5
      </label>
      <label class="btn btn-secondary">
        <input type="radio" name="darkdepth" value=6
          id="darkdepth6"> 6
      </label>
      <label class="btn btn-secondary">
        <input type="radio" name="darkdepth" value=7
          id="darkdepth7"> 7
      </label>
      <label class="btn btn-secondary">
        <input type="radio" name="darkdepth" value=8
          id="darkdepth8"> 8
      </label>
      <label class="btn btn-secondary">
        <input type="radio" name="darkdepth" value=9
          id="darkdepth9"> 9
      </label>
```

```
    </div>
  </div>
</div>

<h6>Light Player Select</h6>
<div class="row">
  <div class="col-12">
    <div class="btn-group btn-group-toggle" data-toggle="buttons"
      id="lightPlayerSelector">
      <label class="btn btn-secondary active">
        <input type="radio" name="darkdepth" value="human"
          id="lightHumanPlayer" checked>
        Human
      </label>
      <label class="btn btn-secondary">
        <input type="radio" name="darkdepth" value=alphabeta
          id="lightCompPlayer">
        Computer
      </label>
    </div>
  </div>
</div>

<h6>Light Depth Select</h6>
<div class="row">
  <div class="col-12">
    <div class="btn-group btn-group-toggle" data-toggle="buttons"
      id="lightdepthSelector">
      <label class="btn btn-secondary active">
        <input type="radio" name="lightdepth" value=1
          id="lightdepth1" checked> 1
      </label>
      <label class="btn btn-secondary">
        <input type="radio" name="lightdepth" value=2
          id="lightdepth2"> 2
      </label>
      <label class="btn btn-secondary">
        <input type="radio" name="lightdepth" value=3
          id="lightdepth3"> 3
      </label>
      <label class="btn btn-secondary">
        <input type="radio" name="lightdepth" value=4
          id="lightdepth4"> 4
      </label>
      <label class="btn btn-secondary">
        <input type="radio" name="lightdepth" value=5
          id="lightdepth5"> 5
      </label>
```

```
                <label class="btn btn-secondary">
                  <input type="radio" name="lightdepth" value=6
                    id="lightdepth6"> 6
                </label>
                <label class="btn btn-secondary">
                  <input type="radio" name="lightdepth" value=7
                    id="lightdepth7"> 7
                </label>
                <label class="btn btn-secondary">
                  <input type="radio" name="lightdepth" value=8
                    id="lightdepth8"> 8
                </label>
                <label class="btn btn-secondary">
                  <input type="radio" name="lightdepth" value=9
                    id="lightdepth9"> 9
                </label>
              </div>
            </div>
          </div>
        </div>
      </div>
      <div class="modal-footer" id="config">
        <button type="button" class="btn btn-secondary"
          data-dismiss="modal">Close</button>
        <button type="button" class="btn btn-primary" data-dismiss="modal"
          id="startButton">Start
          Game</button>
      </div>
    </div>
  </div>
</div>

<div class="modal fade" id="postGameModal" tabindex="-1" role="dialog">
  <div class="modal-dialog modal-dialog-centered modal-lg" role="document">
    <div class="modal-content">
      <div class="modal-header">
        <h5 class="modal-title" id="exampleModalLongTitle">Post
          Game</h5>
        <button type="button" class="close" data-dismiss="modal"
          aria-label="Close">
          <span aria-hidden="true">&times;</span>
        </button>
      </div>
      <div class="modal-body">
        <span id="gameReport"></span>
        <table id="analysisTable">
          <tr>
            <th>Move #</th>
```

```
            <th>Move Played</th>
            <th>Position Evaluation</th>
            <th>Computer Move</th>
            <th>Appraisal</th>
          </tr>
        </table>
      </div>
      <div class="modal-footer">
        <button type="button" class="btn btn-secondary"
          data-dismiss="modal">Close</button>
      </div>
    </div>
  </div>
</div>


<div class="container-fluid">
  <div class="row">
    <div class="col-xl-3">
    </div>
    <div class="col-lg-6 col-12">
      <canvas id="myCanvas" width="500" height="500">
        Your browser does not support the canvas element.
      </canvas>
      <div class="alert alert-info" id="passNotification" role="alert">
        <span id="passLabel">Dark is forced to pass</span>
      </div>

    </div>
    <div class="col-12 col-lg-6 col-xl-3">
      <div id="infoPanel" class="container">
        <div id="configBar" class="container">
          <div class="row">
            <div class="col">
              <button type="button" class="btn btn-primary center-block"
                data-toggle="modal" id="gameSetupBtn"
                data-target="#setupModal">
                New Game
              </button>
            </div>
            <div class="w-100"></div>
            <div class="col">
              <p id="openingLabel">Opening</p>
            </div>
          </div>
        </div>
        <div id="tableWrapper">
          <table id="moveHistory">
          </table>
```

```
          </div>
        </div>
      </div>
    </div>

    <script src="js/reversi.js"></script>
    <script src="js/graphics.js"></script>
    <script src="js/board.js"></script>
    <script src="js/engine.js"></script>
    <script src="js/analyst.js"></script>
    <script src="js/main.js"></script>
    <script src="js/thinker.js"></script>

    <script src="https://code.jquery.com/jquery-3.3.1.slim.min.js"
      integrity="sha384-
q8i/X+965DzO0rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo"
      crossorigin="anonymous"></script>
    <script
      src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd/popper.
min.js"
      integrity="sha384-
UO2eT0CpHqdSJQ6hJty5KVphtPhzWj9WO1clHTMGa3JDZwrnQq4sF86dIHNDz0W1"
      crossorigin="anonymous"></script>
    <script
      src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min
.js"
      integrity="sha384-
JjSmVgyd0p3pXB1rRibZUAYoIIy6OrQ6VrjIEaFf/nJGzIxFDsf4x0xIM+B07jRM"
      crossorigin="anonymous"></script>
</body>
</html>
```

main.css

```
.container-fluid {
  padding: 2rem;
}

p {
  color: white;
}

body {
  background-color: #cccccc;
}

canvas {
```

```
  background: rgb(27, 138, 24);
  margin: auto;
  padding: 0;
  display: block;
  border-radius: 5px;
  margin: solid dark 100px;
}

#moveHistory {
  width: 100%;
  font-size: 1rem;
  font-family: "Lucida Console";
}

#moveHistory tr {
  border-bottom: solid #a2a3bb 1px;
  height: 50px;
  width: 50%;
}

#moveHistory td:nth-child(even),
#moveHistory th:nth-child(even) {
  color: #fff;
  text-align: center;
  width: 50%;
}

th {
  text-align: left;
}

#moveHistory td:nth-child(odd),
#moveHistory th:nth-child(odd) {
  color: #000;
  text-align: center;
  width: 50%;
}

#tableWrapper {
  max-width: 300px;
  height: 400px;
  overflow: scroll;
  background: rgb(27, 138, 24);
  display: block;
  margin: auto;
  border-radius: 5px;
}
```

```
::-webkit-scrollbar {
  display: none;
}

.navbar {
  display: none;
}

#configPanel {
  padding: 20px;
}

@media only screen and (max-width: 1200px) {
  #configPanel {
    margin-bottom: 20px;
  }
}

@media only screen and (max-width: 992px) {
  canvas {
    margin-bottom: 20px;
  }
}

#startButton {
  display: inline;
  margin-right: 5px;
  color: white;
}

#configPanel {
  visibility: hidden;
}

.btn-group {
  width: 100%;
  padding-left: 10px;
  padding-right: 10px;
  padding-top: 30px;
  padding-bottom: 30px;
}

#gameSetupBtn {
  margin: 10px;
  height: 50px;
}

#configBar {
```

```
  background-color: rgba(0, 0, 0, 0.1);
  display: flex;
  justify-content: center;
  height: 100px;
}

#infoPanel {
  height: 500px;
  max-width: 300px;
  background: rgb(27, 138, 24);
  padding: 0;
  display: block;
  margin: auto;
  border-radius: 5px;
}

#configBar .row {
  text-align: center;
}

#openingLabel {
  display: inline-block;
}

#scoreDisplay {
  color: black;
  font-size: 20px;
  display: inline-block;
}

#scoreWrapper {
  text-align: center;
  padding-bottom: 30px;
}

#analysisTable {
  width: 100%;
  margin: 30px 0;
}

#analysisTable th,
#analysisTable td {
  padding: 10px;
}

#analysisTable tr:nth-child(even) {
  background-color: rgb(200, 200, 200);
}
```

```
#passNotification {
  width: 300px;
  margin: 20px auto;
  visibility: hidden;
}
```

main.js

```
$(function() {
  // when the page loads
  console.log("ready");
  engine = new Engine(8);
  engine.draw();
  $("#passNotification").hide();
  document.getElementById("passNotification").style.visibility = "visible";
});

document.getElementById("gameSetupBtn").addEventListener("click", function() {
  // when the new game button is clicked
  engine = null;
  engine = new Engine(8);
  engine.draw();
});

document.getElementById("startButton").addEventListener("click", function() {
  // when the start game button is clicked
  engine.display.showLegals(engine);
  $("#myCanvas").off("mousedown");
  console.log("始めましょう");
  engine.handleTurn();
});
```

reversi.js

```
class Reversi {
  constructor(dims) {
    this.board = new Board(dims);
    this.gameRunning = true;
    this.darkTurn = true;
    this.missedTurns = 0;
    this.freeTiles = dims ** 2 - 4;
    this.gameHistory = "";
    this.inBook = true;
    Object.defineProperty(this, "openings", {
```

```
      configurable: true,
      writable: true,
      value: []
    });
  }

  turnToken() {
    // returns the token ID of the player whose turn
    // it currently is

    if (this.darkTurn) return 1;
    else return 2;
  }

  gameEnded(field) {
    // determines whether the game has ended by checking
    // whether either player has legal moves available

    const noDarkMoves = this.findLegalMoves(field, 1) == false;
    const noLightMoves = this.findLegalMoves(field, 2) == false;
    return noDarkMoves && noLightMoves;
  }

  insertToken(cell) {
    // checks to see whether a move is legal and, if so
    // inserts the relevant token into the board array

    const [row, col] = cell;
    var success;
    if (JSON.stringify(this.findLegalMoves(this.board.field)).includes(cell))
{
      // if the move is in the list of legal moves
      success = true;
      // insert the token into the board matrix
      this.board.field[row][col] = this.turnToken();
    } else {
      success = false;
    }
    return success;
  }

  findLegalMoves(field, player = null) {
    // finds all legal moves for a given player
    // given some position 'field'

    if (player == null) player = this.turnToken();
    const neighbours = this.findNeighbours(field);
    const legals = this.validateNeighbours(neighbours, field, player);
```

```
  return legals;
}

findNeighbours(field) {
  // finds all squares adjacent to occupied squares

  const occupiedCells = this.findOccupiedCells(field);
  const surrounds = this.generateSurroundVectors();
  var neighbours = [];
  for (let cell of occupiedCells) {
    for (let surround of surrounds) {
      let neighbour = cell.map(function(a, b) {
        return a + surround[b];
      });
      if (
        this.onBoard(neighbour) &&
        field[neighbour[0]][neighbour[1]] == 0
      ) {
        neighbours.push(neighbour);
      }
    }
  }
  var unduplicatedNeighbours = this.deleteDuplicates(neighbours);
  return unduplicatedNeighbours;
}

findOccupiedCells(field) {
  // returns a list of all cells which are occupied

  var occupiedCells = [];
  for (let i = 0; i < field.length; i++) {
    for (let j = 0; j < field[i].length; j++) {
      if (field[i][j] != 0) {
        occupiedCells.push([i, j]);
      }
    }
  }
  return occupiedCells;
}

generateSurroundVectors() {
  // returns a list of vectors corresponding to
  // transformations which map a cell onto one of
  // its neighbouring cells

  var surrounds = [];
  for (let i = -1; i < 2; i++) {
    for (let j = -1; j < 2; j++) {
```

```
      if (i != 0 || j != 0) {
        surrounds.push([i, j]);
      }
    }
  }
  return surrounds;
}

validateNeighbours(neighbours, field, player) {
  // tests whether each 'neighbour' square
  // is actually a legal move for a given player

  var legals = [];
  for (let move of neighbours) {
    var runs = this.findRuns(field, move, player);
    if (runs.length > 0) {
      legals.push(move);
    }
  }
  return legals;
}

nextCellInRun(cell, direction) {
  // given a cell, finds the adjacent cell
  // in a given direction

  return cell.map(function(x, y) {
    return x + direction[y];
  });
}

deleteDuplicates(legals) {
  // deletes duplicate moves in a list of
  // legal moves

  var result = [];
  for (let move of legals) {
    if (!JSON.stringify(result).includes(move)) {
      result.push(move);
    }
  }
  return result;
}

onBoard(cell) {
  // tests whether a square is on the board
  // i.e. [3, 4] is on the board but
  // [9, -3] is not (in 8x8)
```

```
  var valid = true;
  for (var coord of cell) {
    valid = valid && 0 <= coord && coord < this.board.cols;
  }
  return valid;
}

findRuns(field, cell, playerToken) {
  // generates a list of 'runs' along which
  // discs must be flipped as a result of a
  // move

  var directions = this.generateSurroundVectors();
  var runs = [];
  for (let direction of directions) {
    var run = [];
    var testedCell = cell;
    while (true) {
      testedCell = this.nextCellInRun(testedCell, direction);
      if (!this.onBoard(testedCell)) {
        break;
      }
      var testedToken = this.tokenAt(testedCell, field);
      if (testedToken == 0) {
        break;
      } else if (testedToken == playerToken) {
        if (run.length > 0) {
          runs.push(run);
        }
        break;
      } else if (testedToken == 3 - playerToken) {
        run.push(testedCell);
      }
    }
  }
  return runs;
}

flipTokens(field, cell, token) {
  // flips the necessary discs given a move (cell)
  // that has just been played and a board position (field)

  var runs = this.findRuns(field, cell, token);
  for (var run of runs) {
    for (var cell of run) {
      field[cell[0]][cell[1]] = token;
    }
```

```
  }
}

countTokens(board) {
  // counts the number of discs for
  // each player

  var dark = 0,
    light = 0;
  for (var i = 0; i < board.rows; i++) {
    for (var j = 0; j < board.cols; j++) {
      if (board.field[i][j] == 1) dark++;
      else if (board.field[i][j] == 2) light++;
    }
  }
  return [dark, light];
}

handlePass() {
  // is called whenever a player is
  // forced to pass (because they have no
  // legal moves)

  this.missedTurns += 1;
  this.darkTurn = !this.darkTurn;
  this.gameHistory += "--";
}

getRandomMove() {
  // returns a random, legal move

  var legals = this.findLegalMoves(this.board.field);
  var move = legals[Math.floor(Math.random() * legals.length)];
  return move;
}

loadOpenings(callback) {
  // requests the opening book from a text file

  var self = this;
  let xhttp = new XMLHttpRequest();
  xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
      callback(this, self);
    }
  };
  xhttp.open("GET", "books/openings.txt", true);
  xhttp.send();
```

```
}

callback(xhttp, self) {
  // is run when the opening book text file is posted

  let openings = xhttp.responseText.split("\n");
  for (let i = 0; i < openings.length; i++) {
    openings[i] = openings[i].toLowerCase().split(", ");
  }
  self.openings = openings;
}

checkInBook() {
  // checks whether the game is still following
  // a known opening and updates this.inBook accordingly.
  // returns a list of possible openings that are consistent
  // with the game so far

  var possibleOpenings = this.findPossibleOpenings(this.gameHistory);
  var viableOpenings = this.filterLongerOpenings(possibleOpenings);
  if (viableOpenings.length > 0) {
    this.inBook = true;
  } else {
    this.inBook = false;
  }
  return viableOpenings;
}

getBookMove(openings) {
  // randomly chooses a move from a book opening
  // that is consistent with the game so far

  let progress = this.gameHistory.length;
  let opening = openings[Math.floor(Math.random() * openings.length)];
  let permutedMove = this.machineForm(
    opening[0][0].slice(progress, progress + 2).split("")
  );
  var move = this.board.permuteCell(permutedMove, opening[1]);
  move = [move[0], move[1]];
  return move;
}

findPossibleOpenings(transcript) {
  // returns a list of all book openings
  // that are consistent with the game so far

  let possibleOpenings = [];
  for (let permutation = 0; permutation < 4; permutation++) {
```

```
    var perumtedHistory = "";
    for (let i = 0; i < transcript.length / 2; i++) {
      let cell = this.machineForm(
        transcript.slice(i * 2, i * 2 + 2).split("")
      );
      let permutedCell = this.humanForm(
        this.board.permuteCell(cell, permutation)
      ).join("");
      perumtedHistory += permutedCell;
    }
    for (let opening of this.openings) {
      if (opening[0].startsWith(perumtedHistory)) {
        possibleOpenings.push([opening, permutation]);
      }
    }
  }
  return possibleOpenings;
}

filterLongerOpenings(openings) {
  // filters a list of openings to only
  // include openings that contain moves
  // beyond the current game

  let longerOpenings = [];
  for (let opening of openings) {
    if (opening[0][0].length > this.gameHistory.length) {
      longerOpenings.push(opening);
    }
  }
  return longerOpenings;
}

getCompletedOpening(openings) {
  // returns the longest opening that has been
  // fully played out in the current game

  let completedOpening = [[""]];
  for (let opening of openings) {
    if (
      opening[0][0].length <= this.gameHistory.length &&
      opening[0][0].length > completedOpening[0][0].length
    ) {
      completedOpening = opening;
    }
  }
  return completedOpening;
}
```

```
  machineForm(cell) {
    // converts a coordinate in the form
    // letter, number (e.g. e6) to the corresponding
    // indices of the field matrix (e.g. [4,5])

    let col = cell[0].charCodeAt(0) - 97;
    let row = cell[1] - 1;
    return [row, col];
  }

  humanForm(cell) {
    // the inverse function of machineForm().

    var col = String.fromCharCode(cell[1] + 97);
    var row = cell[0] + 1;
    return [col, row];
  }

  mockPlay(field, move, token) {
    // returns the board position that results from
    // inserting a given token at a given position from
    // some specified position

    if (move == null) return field;
    let newfield = JSON.parse(JSON.stringify(field));
    newfield[move[0]][move[1]] = token;
    this.flipTokens(newfield, [move[0], move[1]], token);
    return newfield;
  }

  tokenAt(cell, field) {
    // returns the token (0, 1, or 2) at a certain
    // square in the field matrix

    var y = cell[0];
    var x = cell[1];
    var token = field[y][x];
    return token;
  }
}
```

engine.js

```
class Engine extends Reversi {
  constructor(dims) {
    super(dims);
    this.display = new Display(dims);
```

```
  this.darkPlayer = null;
  this.lightPlayer = null;
  this.darkDepth = 0;
  this.lightDepth = 0;
  this.aiDelay = 0; //miliseconds
  this.useBook = true;
  this.thinking = false;
  this.initGameTable();
  this.initDepthSelect();
  this.initPlayerSelect();
  this.initOpeningLabel();
  this.initAnalysisTable();
  this.loadOpenings(this.callback);
}

initDepthSelect() {
  // sets the depth select radio buttons to
  // update the relevant depth attribute

  var self = this;
  this.darkDepth = 1;
  console.log("intitialising depth selection");
  var darkselect = $("#darkdepthSelector .btn");
  darkselect[0].click();
  darkselect.on("click", function(event) {
    let darkdepth = $(this)
      .find("input")
      .val();
    self.darkDepth = Number(darkdepth);
  });
  this.lightDepth = 1;
  var lightselect = $("#lightdepthSelector .btn");
  lightselect[0].click();
  lightselect.on("click", function(event) {
    let lightdepth = $(this)
      .find("input")
      .val();
    self.lightDepth = Number(lightdepth);
  });
}

initPlayerSelect() {
  // sets the player select radio buttons to
  // update the relevant player attribute

  var self = this;
  this.darkPlayer = "human";
  console.log("intitialising player selection");
```

```
  var darkselect = $("#darkPlayerSelector .btn");
  darkselect[0].click();
  darkselect.on("click", function(event) {
    let darkplayer = $(this)
      .find("input")
      .val();
    self.darkPlayer = darkplayer;
  });
  this.lightPlayer = "human";
  var lightselect = $("#lightPlayerSelector .btn");
  lightselect[0].click();
  lightselect.on("click", function(event) {
    let lightplayer = $(this)
      .find("input")
      .val();
    self.lightPlayer = lightplayer;
  });
}

updateOpeningLabel() {
  // updates the label displaying the opening being played

  let label = document.getElementById("openingLabel");
  let possibleOpenings = this.findPossibleOpenings(this.gameHistory);
  let currentOpening = this.getCompletedOpening(possibleOpenings);
  if (typeof currentOpening[0][1] == "string") {
    label.innerHTML = currentOpening[0][1];
  }
}

updateGameTable() {
  // updates the game table after a move is played

  let table = document.getElementById("moveHistory");
  if (this.gameHistory.length % 4 === 0) {
    console.log(table.rows[0]);
    let row = table.rows[table.rows.length - 1];
    let cell = row.cells[1];
    console.log(row);
    cell.innerHTML = this.gameHistory.slice(-2);
  } else {
    let row = table.insertRow(-1);
    let cell1 = row.insertCell(0);
    cell1.innerHTML = this.gameHistory.slice(-2);
    row.insertCell(1);
  }
}
```

```
initGameTable() {
  // clears game table ready for new game

  let table = document.getElementById("moveHistory");
  let len = table.rows.length;
  for (let i = 0; i < len; i++) {
    table.deleteRow(0);
  }
}

initAnalysisTable() {
  var table = document.getElementById("analysisTable");
  var len = table.rows.length;
  for (let i = 0; i < len - 1; i++) {
    table.deleteRow(1);
  }
}

initOpeningLabel() {
  //initialises label displaying opening being played

  let label = document.getElementById("openingLabel");
  label.innerText = "Opening";
}

draw() {
  // wrapper for the drawBoard() method

  this.display.drawBoard(this.board);
}

handleTurn() {
  // is called every turn and handles
  // the logic of a single turn

  if (this.gameEnded(this.board.field)) {
    this.gameRunning = false;
    this.handlegameEnd();
  } else {
    // if game is still running
    if (this.findLegalMoves(this.board.field) == false) {
      this.handlePass();
      $("#passNotification").show();
      var passLabel = document.getElementById("passLabel");
      var playerText = this.darkTurn ? "Light" : "Dark";
      passLabel.innerText = playerText + " is forced to pass";
      this.updateGameTable();
      this.display.showLegals(this);
```

```
      var self = this;
      setTimeout(function() {
        $("#passNotification").hide();
        self.handleTurn(); //
      }, 1000);
    } else {
      // if there are legal moves available
      var activePlayer = this.darkTurn ? this.darkPlayer : this.lightPlayer;
      console.log("activeplayer: ", activePlayer);
      if (activePlayer === "human") {
        var self = this;
        $("#myCanvas").one("mousedown", function() {
          self.processHumanMove();
        });
        console.log("listening for input");
      } else if (!this.thinking) {
        //if computer player
        var self = this;
        this.thinking = true;
        console.log("processing CompMove");
        setTimeout(function() {
          self.processCompMove();
        }, this.aiDelay);
      }
    }
  }
}

initiateMinimax() {
  // constructs a list of parameters to be
  // passed to the minimax worker and then
  // initiates the creation of the worker

  var depth = this.darkTurn ? this.darkDepth : this.lightDepth;
  if (this.freeTiles <= 10) {
    depth = 10;
  }
  var fieldCopy = JSON.parse(JSON.stringify(this.board.field));
  var workerMessage = [
    depth,
    fieldCopy,
    this.turnToken(),
    this.freeTiles,
    this.board.rows
  ];
  this.createMinimaxWorker(workerMessage);
}
```

```
createMinimaxWorker(workerMessage) {
  // creates a minimax web worker with a
  // message. This worker asynchronously
  // performas the minimax algorithm

  var self = this;
  var w = new Worker("js/minimaxWorker.js");
  w.postMessage(workerMessage);
  w.onmessage = function(event) {
    var result = event.data;
    var move = result.move;
    if (typeof self != "undefined") {
      self.executeMove(move);
    }
    w.terminate(); // delete web worker
    w = undefined; // and remove the reference
  };
}

processCompMove() {
  // is called by handleTurn() if it is the
  // computers turn to move and handles the process
  // of deciding on a computer move

  var executed = false;
  if (this.gameHistory == "") {
    var move = this.getRandomMove();
    executed = true;
    this.executeMove(move);
  } else if (this.inBook && this.useBook) {
    var openings = this.checkInBook();
    if (this.inBook) {
      var move = this.getBookMove(openings);
      executed = true;
      this.executeMove(move);
    }
  }
  if (!executed) {
    this.initiateMinimax();
  }
}

executeMove(move) {
  // handles the execution of a move and updates
  // various game related attributes accordingly

  console.log("executing", move);
  if (this.insertToken(move)) {
```

```
    this.flipTokens(this.board.field, move, this.turnToken());
    this.display.ctx.clearRect(
      0,
      0,
      this.display.canvas.width,
      this.display.canvas.height
    );
    this.draw();
    this.darkTurn = !this.darkTurn;
    this.display.showLegals(this);
    this.gameHistory += this.humanForm(move).join("");
    console.log(this.gameHistory);
    this.display.highlightLastSquare(this);
    this.updateGameTable();
    this.updateOpeningLabel();
    this.freeTiles -= 1;
    this.thinking = false;
  }
  this.handleTurn();
}

processHumanMove() {
  // processes human input (clicking on the board)
  // and inputs the corresponding move, updating
  // various game attributes accordingly

  let move = this.getSquare(event);
  this.executeMove(move);
}

getSquare(event) {
  // determines which square has been clicked
  // given the coordinates of a mouseclick on
  // the board

  const bound = this.display.canvas.getBoundingClientRect();
  var x = event.clientX - bound.left;
  var y = event.clientY - bound.top;
  var cellX = Math.floor(x / this.display.colWidth),
    cellY = Math.floor(y / this.display.rowHeight);
  return [cellY, cellX];
}

handlegameEnd() {
  // determines the winner at the end of
  // the game and triggers the post-game
  // popup window
```

```
  var [dark, light] = this.countTokens(this.board);
  console.log("dark", dark);
  console.log("light", light);
  var prefix;
  if (dark > light) {
    prefix = "Dark Won ";
  } else if (light > dark) {
    prefix = "Light Won ";
  } else {
    prefix = "Draw: ";
  }
  let scoreline = prefix + dark + " - " + light;
  let scoreLabel = document.getElementById("gameReport");
  scoreLabel.innerText = scoreline;
  $("#postGameModal").modal("show");
  this.analyseGame(3);
}

analyseGame(depth) {
  // analyses the game just played and constructs
  // an analysis in memory

  var analyst = new Analyst(this.board.rows, this.gameHistory);
  var evaluations = analyst.gradePositions(depth);
  var self = this;
  evaluations.then(function(results) {
    var tableData = analyst.constructAnalysisTable(results);
    var humanisedTableData = [];
    for (var row of tableData) {
      try {
        row[3] = self.humanForm(row[3]).join("");
      } catch {
        row[3] = "N/A";
      }
      humanisedTableData.push(row);
    }
    self.populteAnalysisTable(humanisedTableData);
  });
}

populteAnalysisTable(tableData) {
  // populates an HTML table with the contents
  // of a precontructed analysis table

  var table = document.getElementById("analysisTable");
  for (let rowData of tableData) {
    var row = table.insertRow();
    for (let cellData of rowData) {
```

```
        var cell = row.insertCell();
        cell.innerHTML = cellData;
      }
    }
  }
}
```

board.js

```
class Board {
  constructor(dims) {
    this.rows = this.cols = dims;
    this.field = this.constructStartingBoard();
  }

  constructStartingBoard() {
    // constructs a board matrix representing
    // the starting position

    var field = [];
    for (var i = 0; i < this.rows; i++) {
      field.push([]);
      for (var j = 0; j < this.cols; j++) {
        field[i].push(0);
      }
    }
    let half = this.rows / 2;
    field[half - 1][half - 1] = field[half][half] = 2;
    field[half][half - 1] = field[half - 1][half] = 1;
    return field;
  }

  rotateCell(cell, turns) {
    // rotates a cell by a specified amount about the centre
    // 'turns' is the number of quarter turns anticlockwise

    var newCell = cell;
    for (let i = 0; i < turns; i++) {
      let col = this.rows - 1 - newCell[1];
      let row = newCell[0];
      newCell = [col, row];
    }
    return newCell;
  }

  reflectCell(cell, direction) {
    // reflects a cell in one of the boards diagonals
    // direction = 0 refers to the leading diagonal
```

```
    // direction = 1 refers to the trailing diagonal

    var newCell;
    if (direction == 0) {
      newCell = cell.reverse();
    } else {
      newCell = [this.rows - 1 - cell[1], this.cols - 1 - cell[0]];
    }
    return newCell;
  }

  permuteCell(cell, permutation) {
    // performs one of the four permutations to a cell
    // permutation = 0 => nothing (the 'identity')
    // permutation = 1 => reflection in leading diagonal
    // permutation = 2 => rotation by half turn about centre
    // permutation = 3 => reflection in trailing diagonal

    var newCell;
    switch (permutation) {
      case 0:
        newCell = cell;
        break;
      case 1:
        newCell = this.reflectCell(cell, 0);
        break;
      case 2:
        newCell = this.rotateCell(cell, 2);
        break;
      case 3:
        newCell = this.reflectCell(cell, 1);
        break;
    }
    return newCell;
  }
}
```

graphics.js

```
class Display {
  constructor(dims) {
    this.canvas = document.getElementById("myCanvas");
    this.ctx = this.canvas.getContext("2d");
    this.cols = dims;
    this.rows = dims;
    this.colWidth = this.canvas.width / this.cols;
    this.rowHeight = this.canvas.height / this.rows;
    this.darkColour = "#000000";
```

```
  this.lightColour = "#ffffff";
}

drawBoard(board) {
  // draws the board on the HTML Canvas object

  this.drawGrid();
  this.labelSquares();
  this.drawDiscs(board);
}

drawGrid() {
  // draws the grid seperating each square on
  // the HTML canvas

  this.ctx.clearRect(0, 0, this.canvas.width, this.canvas.height);
  this.ctx.strokeStyle = "#A2A3BB";
  for (var i = 1; i < this.rows; ++i) {
    this.ctx.beginPath();
    this.ctx.moveTo(this.colWidth * i, 0);
    this.ctx.lineTo(this.colWidth * i, this.canvas.height);
    this.ctx.stroke();
    this.ctx.closePath();

    this.ctx.beginPath();
    this.ctx.moveTo(0, this.rowHeight * i);
    this.ctx.lineTo(this.canvas.width, this.rowHeight * i);
    this.ctx.stroke();
    this.ctx.closePath();
  }
}

labelSquares() {
  // labels the top and left side of the board
  // with the coordinate system

  for (let i = 0; i < this.rows + 1; i++) {
    this.ctx.font = "14px Lucida Console";
    this.ctx.fillStyle = "#D9C9C9";
    this.ctx.fillText(
      String.fromCharCode(i + 96),
      this.colWidth * i - 14,
      15
    );
    this.ctx.fillText(i, 1, this.rowHeight * i - 10);
  }
}
```

```
drawDiscs(board) {
  // draws each player's discs on the HTML canvas

  for (var i = 0; i < board.field.length; i++) {
    for (var j = 0; j < board.field[i].length; j++) {
      if (board.field[i][j] == 1) {
        this.ctx.fillStyle = this.darkColour;
        var x = this.colWidth * j + this.canvas.width / (this.cols * 2);
        var y = this.rowHeight * i + this.canvas.height / (this.rows * 2);
        this.ctx.beginPath();
        this.ctx.arc(x, y, this.colWidth / 3, 0, 2 * Math.PI);
        this.ctx.fill();
      } else if (board.field[i][j] == 2) {
        this.ctx.fillStyle = this.lightColour;
        x = this.colWidth * j + this.canvas.width / (this.cols * 2);
        y = this.rowHeight * i + this.canvas.height / (this.rows * 2);
        this.ctx.beginPath();
        this.ctx.arc(x, y, this.colWidth / 3, 0, 2 * Math.PI);
        this.ctx.fill();
        this.ctx.closePath();
      }
    }
  }
}

showLegals(game) {
  // displays the legal moves for the active player
  // as small dots on the canvas

  let legals = game.findLegalMoves(game.board.field);
  for (var i = 0; i < game.board.rows; i++) {
    for (var j = 0; j < game.board.cols; j++) {
      var x = this.colWidth * (j + 0.5),
        y = this.rowHeight * (i + 0.5);
      switch (game.turnToken()) {
        case 1:
          this.ctx.fillStyle = this.darkColour;
          break;
        case 2:
          this.ctx.fillStyle = this.lightColour;
          break;
      }
      if (JSON.stringify(legals).includes([i, j])) {
        this.ctx.beginPath();
        this.ctx.arc(x, y, 5, 0, 2 * Math.PI);
        this.ctx.fill();
      }
    }
```

```
    }
  }

  highlightSquare(square) {
    // highlights a given sqaure on the board

    this.ctx.fillStyle = "rgba(0, 0, 0, 0.2)";
    let x = square[1] * this.colWidth;
    let y = square[0] * this.rowHeight;
    this.ctx.fillRect(x, y, this.colWidth, this.rowHeight);
  }

  highlightLastSquare(game) {
    // highlights the square to white a move was
    // most recently played

    let lastSquare = game.machineForm(game.gameHistory.slice(-2));
    this.highlightSquare(lastSquare);
  }
}
```

thinker.js

```
class CompPlayer extends Reversi {
  constructor(dims) {
    super(dims);
  }

  evaluateByWeight(field, maxToken) {
    // applies a weightings mask to a given game
    // position to determine, based on this alone,
    // which player this position is more desirable for

    var weightings = this.generateWeightingMask();
    let result = 0;
    for (let i = 0; i < weightings.length; i++) {
      for (let j = 0; j < weightings[i].length; j++) {
        let mult;
        if (field[i][j] == maxToken) {
          mult = 1;
        } else if (field[i][j] == 3 - maxToken) {
          mult = -1;
        } else {
          mult = 0;
        }
        result += mult * weightings[i][j];
      }
    }
```

```
  return result;
}

generateWeightingMask() {
  // constructs a weighting 'mask' based on which positions are
  // likely to be desirable to occupy

  let weightings = [];
  let n = this.board.rows - 1;
  for (let i = 0; i < n + 1; i++) {
    let row = [];
    for (let j = 0; j < n + 1; j++) {
      var score;
      var cell = [i, j];
      if (this.cornerSquare(cell, this.board.rows)) {
        score = 100;
      } else if (this.cSquare(cell, this.board.rows)) {
        score = -10;
      } else if (this.aOrBSquare(cell, this.board.rows)) {
        score = 0;
      } else if (this.xSquare(cell, this.board.rows)) {
        score = -10;
      } else {
        score = 0;
      }
      row.push(score);
    }
    weightings.push(row);
  }
  return weightings;
}

cornerSquare(cell, dims) {
  // tests whether a given square is a
  // corner square

  var n = dims - 1;
  var [i, j] = cell;
  var x = j - n / 2;
  var y = i - n / 2;
  return x ** 2 + y ** 2 == n ** 2 / 2;
}

cSquare(cell, dims) {
  // tests whether a given square is a
  // C square

  var n = dims - 1;
```

```
  var [i, j] = cell;
  var x = j - n / 2;
  var y = i - n / 2;
  return x ** 2 + y ** 2 == ((n - 1) ** 2 + 1) / 2;
}

aOrBSquare(cell, dims) {
  // tests whether a given square is an
  // A square or B square

  var n = dims - 1;
  var [i, j] = cell;
  return [i, j].map(x => [0, n].includes(x)).reduce((x, y) => x || y);
}

xSquare(cell, dims) {
  // tests whether a given square is an
  // X square

  var n = dims - 1;
  var [i, j] = cell;
  var x = j - n / 2;
  var y = i - n / 2;
  return x ** 2 + y ** 2 == (n - 2) ** 2 / 2;
}

countFrontiers(field) {
  // returns the number of frontier discs for light and dark

  let dark = 0;
  let light = 0;
  const surrounds = this.generateSurroundVectors();
  for (let i = 0; i < field.length; i++) {
    for (let j = 0; j < field[i].length; j++) {
      if (field[i][j] != 0) {
        let frontier = false;
        for (let s of surrounds) {
          let cell = [i + s[0], j + s[1]];
          if (this.onBoard(cell) && field[cell[0]][cell[1]] == 0) {
            frontier = true;
          }
        }
        if (frontier) {
          if (field[i][j] == 1) {
            dark++;
          } else {
            light++;
          }
```

```
        }
      }
    }
  }
  return [dark, light];
}

evaluateByTerritory(field, maxToken) {
  // evuluates the desirability of a position for a given
  // player based only on how many discs each player has

  let dark = 0,
    light = 0;
  for (let i = 0; i < field.length; i++) {
    for (let j = 0; j < field[i].length; j++) {
      if (field[i][j] == 1) dark++;
      else if (field[i][j] == 2) light++;
    }
  }
  return maxToken == 1 ? dark - light : light - dark;
}

evaluateByMobility(field, maxToken) {
  // evaluates the desirability of a position for a given
  // player based only on each player's mobility

  const darkMoves = this.findLegalMoves(field, 1).length;
  const lightMoves = this.findLegalMoves(field, 2).length;
  const team = maxToken == 1 ? 1 : -1;
  const mobility = darkMoves - lightMoves;
  return mobility * team;
}

evaluateByFrontiers(field, maxToken) {
  // evaluates the desirability of a position for a given
  // player based only on the number of frontier discs possessed
  // by each player

  const frontiers = this.countFrontiers(field);
  const [dark, light] = frontiers;
  return maxToken == 1 ? light - dark : dark - light;
}

evaluate(field, maxToken, tilesLeft) {
  // evaluates how desirable some position is for a given
  // player based on an amalgamation of various metrics

  if (this.gameEnded(field)) {
```

```
    var score = this.evaluateByTerritory(field, maxToken);
    if (score > 0) {
      return 1000 + score;
    } else if (score < 0) {
      return -1000 + score;
    } else {
      return 0;
    }
  }
  if (tilesLeft > 5) {
    const mc = 1,
      fc = 1,
      ec = 1;
    const mobility = this.evaluateByMobility(field, maxToken);
    const frontiers = this.evaluateByFrontiers(field, maxToken);
    const edges = this.evaluateByWeight(field, maxToken);
    return mc * mobility + fc * frontiers + ec * edges;
  } else return this.evaluateByTerritory(field, maxToken);
}

inv(move) {
  // returns a reversed copy of a move index

  if (move == null) {
    return move;
  } else {
    return [move[1], move[0]];
  }
}

maxMove(a, b) {
  // returns the ValuedMove object with the
  // greatest value attribute

  let vals = [];
  for (let m of [a, b]) {
    while (m instanceof ValuedMove) {
      m = m.value;
    }
    vals.push(m);
  }
  return vals[0] >= vals[1] ? a : b;
}

minMove(a, b) {
  // returns the ValuedMove object with the
  // least value
```

```
  let vals = [];
  for (let m of [a, b]) {
    while (m instanceof ValuedMove) {
      m = m.value;
    }
    vals.push(m);
  }
  return vals[0] <= vals[1] ? a : b;
}

alphabeta(
  field,
  depth,
  maxToken,
  tilesLeft,
  alpha = -10000,
  beta = 10000,
  maximisingPlayer = true
) {
  // searches the game tree to a specified depth
  // and returns the best move from some position

  if (depth === 0 || this.gameEnded(field)) {
    var evaluation = this.evaluate(field, maxToken, tilesLeft);
    return new ValuedMove(evaluation, null);
  }
  let simToken = maximisingPlayer ? maxToken : 3 - maxToken;
  let legals = this.findLegalMoves(field, simToken);
  if (legals == false) legals.push(null);
  if (maximisingPlayer) {
    var value = new ValuedMove(-10000, null);
    for (var move of legals) {
      var newfield = this.mockPlay(field, move, simToken);
      var newValue = this.alphabeta(
        newfield,
        depth - 1,
        maxToken,
        tilesLeft,
        alpha,
        beta,
        false
      ).value;
      var newValuedMove = new ValuedMove(newValue, move);
      value = this.maxMove(value, newValuedMove);
      alpha = this.maxMove(alpha, value.value);
      if (alpha >= beta) break;
    }
    return value;
```

```
    } else {
      // minimising player
      var value = new ValuedMove(10000, null);
      for (var move of legals) {
        var newfield = this.mockPlay(field, move, simToken);
        var newValue = this.alphabeta(
          newfield,
          depth - 1,
          maxToken,
          tilesLeft,
          alpha,
          beta,
          true
        ).value;
        var newValuedMove = new ValuedMove(newValue, move);
        value = this.minMove(value, newValuedMove);
        beta = this.minMove(beta, value.value);
        if (alpha >= beta) break;
      }
      return value;
    }
  }
}

class ValuedMove {
  constructor(value, move) {
    this.value = value;
    this.move = move;
  }
}
```

analyst.js

```
class Analyst {
  constructor(dims, transcript) {
    this.transcript = transcript;
    this.simulator = new CompPlayer(dims);
    this.simulator.loadOpenings(this.simulator.callback);
    this.positions = this.genPositions();
  }

  genPositions() {
    // returns a list of all positions encountered in a game
    // from a transcript of the game

    var positions = [];
    var field = JSON.parse(JSON.stringify(this.simulator.board.field));
    for (let i = 0; i < this.transcript.length; i += 2) {
```

```
    var move = this.transcript.slice(i, i + 2);
    var token = i % 4 == 0 ? 1 : 2;
    if (move != "--") {
      move = this.simulator.machineForm(move);
      var field = this.simulator.mockPlay(field, move, token);
      positions.push(field);
    } else {
      positions.push(field);
    }
  }
  return positions;
}

gradePositions(depth) {
  // asynchronously computes the value of each position using an
  // alpha-beta search on a given search depth

  var specificDepth = depth;
  var self = this;
  var evaluations = new Array(this.positions.length);
  var freeTiles = this.simulator.board.rows ** 2 - 1;
  for (var i = 0; i < this.positions.length; i++) {
    if (i > 0 && this.positions[i] != this.positions[i - 1]) {
      freeTiles -= 1;
    }
    if (freeTiles <= 10) {
      // solves the game if there are 10 or
      // fewer tiles left
      specificDepth = 100;
    }
    var fieldCopy = JSON.parse(JSON.stringify(this.positions[i]));
    var token = i % 2 == 0 ? 2 : 1;
    evaluations[i] = new Promise(function(resolve) {
      var w = new Worker("js/minimaxWorker.js");
      var workerMessge = [
        specificDepth,
        fieldCopy,
        token,
        freeTiles,
        self.simulator.board.rows
      ];
      w.postMessage(workerMessge);
      w.onmessage = function(event) {
        var colour = i % 2 == 0 ? -1 : 1;
        var result = event.data;
        var value = result.value * colour;
        var move = result.move;
        w.terminate();
```

```
        w = undefined;
        resolve([value, move]);
      };
    });
  }
  return Promise.all(evaluations);
}

separateEvaluations(evaluations) {
  // splits the value and best move for each position
  // into seperate lists for values and best moves

  var values = new Array(evaluations.length);
  var moves = new Array(evaluations.length);
  for (var i = 0; i < evaluations.length; i++) {
    values[i] = evaluations[i][0];
    if (i != evaluations.length - 1) {
      moves[i + 1] = evaluations[i][1];
    }
  }
  return [values, moves];
}

constructAnalysisTable(evaluations) {
  // constructs a 2D array table to display in the
  // post game analysis popup

  var [values, moves] = this.separateEvaluations(evaluations);
  var table = [];
  for (let i = 0; i < evaluations.length; i++) {
    var playedMove = this.transcript.slice(i * 2, (i + 1) * 2);
    var appraisal = this.appraiseMove(i, values, moves, playedMove);
    var row = [i + 1, playedMove, values[i], moves[i], appraisal];
    table.push(row);
  }
  return table;
}

appraiseMove(n, values, moves, playedMove) {
  // assigns a qualitative label to each move
  // describing how good it was

  var appraisal = "";
  var valImprovement = values[n + 1] * -1 - values[n];
  var playedOpening = this.determineOpening(this.transcript);
  var openingLength = playedOpening[0].length / 2;
  if (moves[n]) {
    if (n <= openingLength) {
```

```
      appraisal = "book: " + playedOpening[1];
    } else {
      if (playedMove == this.simulator.humanForm(moves[n]).join("")) {
        appraisal = "correct move";
      } else if (valImprovement >= 0) {
        appraisal = "good move";
      } else {
        appraisal = "mistake";
      }
    }
  }
  return appraisal;
}

determineOpening() {
  // determines the longest opening that is consistent with
  // the game being analysed

  var consistentOpenings = this.findConsistentOpenings();
  var playedOpening = ["", null];
  for (var opening of consistentOpenings) {
    if (opening[0].length > playedOpening[0].length) {
      playedOpening = opening;
    }
  }
  return playedOpening;
}

findConsistentOpenings() {
  // finds all openings which are consistent
  // with the game being analysed

  var consistentOpenings = [];
  for (var permutation = 0; permutation < 4; permutation++) {
    var permutedHistory = "";
    for (var i = 0; i < this.transcript.length / 2; i++) {
      var cell = this.simulator.machineForm(
        this.transcript.slice(i * 2, i * 2 + 2).split("")
      );
      var permutedCell = this.simulator
        .humanForm(this.simulator.board.permuteCell(cell, permutation))
        .join("");
      permutedHistory += permutedCell;
    }
    for (var opening of this.simulator.openings) {
      if (permutedHistory.startsWith(opening[0])) {
        consistentOpenings.push(opening);
      }
```

```
      }
    }
    return consistentOpenings;
  }
}
```

minimaxWorker.js

```
importScripts("board.js", "reversi.js", "thinker.js");

self.onmessage = function(msg) {
  let [depth, fieldCopy, turnToken, freeTiles, dims] = msg.data;
  var minimaxer = new CompPlayer(dims);
  result = minimaxer.alphabeta(fieldCopy, depth, turnToken, freeTiles);
  delete minimaxer;
  self.postMessage(result);
};
```

openings.txt

```
C4c3, Diagonal Opening
C4c3D3c5B2, X-square Opening
C4c3D3c5B3, Snake/Peasant
C4c3D3c5B3f3, Lysons
C4c3D3c5B3f4B5b4C6d6F5, Pyramid/Checkerboarding Peasant
C4c3D3c5B4, Heath/Tobidashi
C4c3D3c5B4d2C2f4D6c6F5e6F7, Mimura Variation II
C4c3D3c5B4d2D6, Heath-Bat
C4c3D3c5B4d2E2, Iwasaki Variation
C4c3D3c5B4e3, Heath-Chimney
C4c3D3c5B5, Raccoon Dog
C4c3D3c5B6, Rocket
C4c3D3c5B6c6B5, Hamilton
C4c3D3c5B6e3, Lollipop
C4c3D3c5D6, Cow
C4c3D3c5D6e3, Chimney
C4c3D3c5D6f4B4, Cow Bat/Bat/Cambridge
C4c3D3c5D6f4B4b6B5c6B3, Bat (Piau Continuation 2)
C4c3D3c5D6f4B4b6B5c6F5, Melnikov/Bat (Piau Continuation 1)
C4c3D3c5D6f4B4c6B5b3B6e3C2a4A5a6D2, Bat (Kling Continuation)
C4c3D3c5D6f4B4e3B3, Bat (Kling Alternative)
C4c3D3c5D6f4F5, Rose-v-Toth
C4c3D3c5D6f4F5d2, Tanida
C4c3D3c5D6f4F5d2B5, Aircraft/Feldborg
C4c3D3c5D6f4F5d2G4d7, Sailboat
```

C4c3D3c5D6f4F5e6C6d7, Maruoka
C4c3D3c5D6f4F5e6F6, Landau
C4c3D3c5F6, Buffalo/Kenichi Variation
C4c3D3c5F6e2C6, Maruoka Buffalo
C4c3D3c5F6e3C6f5F4g5, Tanida Buffalo
C4c3D3c5F6f5, Hokuriku Buffalo
C4c3E6c5, Wing Variation
C4c3F5c5, Semi-Wing Variation
C4c5, Parallel Opening
C4e3, Perpendicular Opening
C4e3F4c5D6e6, Mimura
C4e3F4c5D6f3C6, Shaman/Danish
C4e3F4c5D6f3D3, Inoue
C4e3F4c5D6f3D3c3, Iago
C4e3F4c5D6f3E2, Bhagat
C4e3F4c5D6f3E6c3D3e2, Rose
C4e3F4c5D6f3E6c3D3e2B5, Flat
C4e3F4c5D6f3E6c3D3e2B5f5, Rotating Flat
C4e3F4c5D6f3E6c3D3e2B5f5B3, Murakami Variation
C4e3F4c5D6f3E6c3D3e2B5f5B4f6C2e7D2c7, Rotating Flat (Kling Continuation)
C4e3F4c5D6f3E6c3D3e2B6f5, Rose-Birth
C4e3F4c5D6f3E6c3D3e2B6f5B4f6G5d7, Brightstein
C4e3F4c5D6f3E6c3D3e2B6f5G5, Rose-Birdie/Rose-Tamenori
C4e3F4c5D6f3E6c3D3e2B6f5G5f6, Rose-Tamenori-Kling
C4e3F4c5D6f3E6c3D3e2D2, Greenberg/Dawg
C4e3F4c5D6f3E6c6, Ralle
C4e3F4c5E6, Horse
C4e3F5b4, Ganglion/No-Cat
C4e3F5b4F3, Swallow
C4e3F5b4F3f4E2e6G5f6D6c6, No-Cat (Continuation)
C4e3F5e6D3, Italian
C4e3F5e6F4, Cat
C4e3F5e6F4c5D6c6F7f3, Sakaguchi
C4e3F5e6F4c5D6c6F7g5G6, Berner
C4e3F6b4, Bent Ganglion
C4e3F6e6F5, Tiger
C4e3F6e6F5c5C3, Stephenson
C4e3F6e6F5c5C3b4, No-Kung
C4e3F6e6F5c5C3b4D6c6B5a6B6c7, No-Kung (Continuation)
C4e3F6e6F5c5C3c6, Comp'Oth
C4e3F6e6F5c5C3c6D3d2E2b3C1c2B4a3A5b5A6a4A2, F.A.T. Draw
C4e3F6e6F5c5C3c6D6, Lightning Bolt
C4e3F6e6F5c5C3g5, Kung
C4e3F6e6F5c5D3, Leader's Tiger
C4e3F6e6F5c5D6, Brightwell
C4e3F6e6F5c5F4g5G4f3C6d3D6, Ishii
C4e3F6e6F5c5F4g5G4f3C6d3D6b3C3b4E2b6, Mainline Tiger
C4e3F6e6F5c5F4g6F7, Rose-Bill

```
C4e3F6e6F5c5F4g6F7d3, Tamenori
C4e3F6e6F5c5F4g6F7g5, Central Rose-Bill/Dead Draw
C4e3F6e6F5g6, Aubrey/Tanaka
C4e3F6e6F5g6E7c5, Aubrey (Feldborg Continuation)
```

Testing

Tests 1 - 14

Test #	Requirement Tested	Test/Sub-Test Description	Test Data	Expected Output	Actual Output	Result
1	L1	Testing correct behaviour on page load	Loading Webpage	Presented with empty board	Presented with empty board	Pass
2	L2	Testing game setup menu	Clicking on 'new game' button	The user is able to select whether each player will be a human or computer player, as well as selecting the depth to which computer players will search.	The user is prompted to select the players' type and search depth via a series of radio buttons in a popup window	Pass
3	L3	Testing start game button functions correctly	Clicking the 'start game' button from within the game setup menu	The game setup window closes and the game begins.	The game setup window closes and the game begins.	Pass

4	L4	Testing generation of legal moves	Active Player: 1 [0, 0, 0, 0, 0, 0, 0, 0] [0, 0, 0, 0, 0, 0, 0, 0] [0, 0, 0, 0, 0, 0, 0, 0] [0, 0, 0, 0, 0, 0, 0, 0] [0, 0, 0, 0, 0, 0, 0, 0] [0, 0, 0, 0, 0, 0, 0, 0] [0, 0, 0, 0, 0, 0, 0, 0] [0, 0, 0, 0, 0, 0, 0, 0]	[]	[]	Pass
			Active Player: 2 [0, 0, 0, 0, 0, 0, 0, 0] [0, 0, 0, 0, 0, 0, 0, 0] [0, 0, 0, 0, 0, 0, 0, 0] [0, 0, 0, 2, 1, 0, 0, 0] [0, 0, 0, 1, 1, 1, 0, 0] [0, 0, 0, 0, 0, 0, 0, 0] [0, 0, 0, 0, 0, 0, 0, 0] [0, 0, 0, 0, 0, 0, 0, 0]	[[3,5],[5,3],[5,5]]	[[3,5],[5,3],[5,5]]	Pass
			Active Player: 1 [0, 0, 0, 0, 0, 0, 0, 0] [0, 0, 0, 0, 0, 0, 0, 0] [0, 0, 0, 0, 0, 0, 0, 0] [0, 0, 0, 2, 1, 0, 0, 0] [0, 0, 0, 2, 1, 1, 0, 0] [0, 0, 0, 2, 0, 0, 0, 0] [0, 0, 0, 0, 0, 0, 0, 0] [0, 0, 0, 0, 0, 0, 0, 0]	[[2,2],[3,2],[4,2],[5,2],[6,2]]	[[2,2],[3,2],[4,2],[5,2],[6,2]]	Pass
			Active Player: 2 [0, 0, 0, 0, 0, 0, 0, 0] [0, 0, 0, 2, 0, 0, 0, 0] [0, 0, 2, 2, 1, 0, 0, 0] [0, 0, 2, 2, 1, 1, 1, 0] [0, 0, 2, 2, 1, 1, 1, 0] [0, 0, 2, 2, 1, 1, 0, 0] [0, 0, 0, 2, 1, 1, 0, 0] [0, 0, 0, 0, 0, 0, 0, 0]	[[1,5],[2,5],[2,6],[2,7],[3,7],[4,7],[5,6],[5,7],[6,6],[7,5],[7,6]]	[[1,5],[2,5],[2,6],[2,7],[3,7],[4,7],[5,6],[5,7],[6,6],[7,5],[7,6]]	Pass
		Position in which dark must pass but the game is not over	Active Player: 1 [2, 1, 1, 1, 1, 1, 1, 2] [0, 1, 1, 1, 1, 1, 1, 2] [2, 1, 2, 1, 2, 2, 2, 2] [2, 1, 1, 2, 2, 2, 1, 2] [2, 1, 2, 2, 2, 1, 1, 2] [2, 1, 1, 2, 2, 2, 1, 2] [2, 1, 2, 1, 1, 1, 1, 1] [0, 2, 2, 2, 2, 2, 2, 2]	[]	[]	Pass

			Active Player: 2 [2, 1, 1, 1, 1, 1, 1, 2] [2, 2, 2, 2, 2, 2, 2, 2] [2, 2, 2, 1, 2, 2, 2, 2] [2, 1, 2, 2, 2, 2, 1, 2] [2, 1, 2, 2, 2, 1, 1, 2] [2, 1, 1, 2, 2, 2, 1, 2] [2, 1, 2, 1, 1, 1, 1, 1] [0, 2, 2, 2, 2, 2, 2, 2]	[[7,0]]	[[7,0]]	Pass
		Full board	Active Player: 1 [2, 1, 1, 1, 1, 1, 1, 2] [2, 2, 2, 2, 2, 2, 2, 2] [2, 2, 2, 1, 2, 2, 2, 2] [2, 1, 2, 2, 2, 2, 1, 2] [2, 1, 2, 2, 2, 1, 1, 2] [2, 1, 2, 2, 2, 2, 1, 2] [2, 2, 2, 1, 1, 1, 1, 1] [2, 2, 2, 2, 2, 2, 2, 2]	[]	[]	Pass
5	L5	Testing the display of legal moves	Started a game	All of dark's legal moves are correctly displayed as small black dots	All of dark's legal moves are correctly displayed as small black dots	Pass
			Inputted a move for dark	All of light's legal moves are correctly displayed as small white dots	All of light's legal moves are correctly displayed as small white dots	Pass
6	L6a	Testing user clicks to input legal moves are handled correctly	Clicked on a square corresponding to a legal move	The board is redrawn to display a disc of the active player's colour on the clicked square	The board is redrawn to display a disc of the active player's colour on the clicked square	Pass
7	L6b	Testing user clicks to input illegal moves are handled correctly	Clicked on a square corresponding to an illegal move	Nothing happens and the active player does not change	Nothing happens and the active player does not change	Pass

8	L8	Testing if passing is handled correctly	Played a game up to the point at which one player is forced to pass	Message is displayed explaining that the player is forced to pass and the other player becomes the active player	Message is displayed explaining that the player is forced to pass and the other player becomes the active player	Pass
9	L9	Testing whether discs are correctly flipped upon after a move is inputted	Inputted a move for dark	Board is redrawn with trapped light discs correctly flipped to dark	Board is redrawn with trapped light discs correctly flipped to dark	Pass
			Inputted a move for light	Board is redrawn with trapped dark discs correctly flipped to light	Board is redrawn with trapped dark discs correctly flipped to light	Pass
10	L10	Testing whether the game history table is updated after each move	Played d3 as dark	Table is updated and displays d3 in black font in the left column	Table is updated and displays d3 in black font in the left column	Pass
			Played c5 as light	Table is updated and displays c5 in white font in the right column	Table is updated and displays c5 in white font in the right column	Pass
			Arrived at a position in which a player must pass	Table is updated and displays -- in place of the sqaure	Table is updated and displays -- in place of the sqaure	Pass
11	L11	Testing whether the opening	Played C4c3	Label displays 'Diagonal Opening'	Label displays 'Diagonal Opening'	Pass

		label is consistent with the game being played				
			Played C4c3D3c5B2	Label displays 'X-square Opening'	Label displays 'X-square Opening'	Pass
			Played C4c3D3c5B3	Label displays 'Snake/Peasant'	Label displays 'Snake/Peasant'	Pass
			Played C4c3D3c5B3f3	Label displays 'Lysons'	Label displays 'Lysons'	Pass
			Played C4c3D3c5B3f4B5b4C6d6F5	Label displays 'Pyramid/Checkerboarding Peasant'	Label displays 'Pyramid/Checkerboarding Peasant'	Pass
			Played C4c3D3c5B4	Label displays 'Heath/Tobidashi'	Label displays 'Heath/Tobidashi'	Pass
			Played C4c3D3c5B4d2C2f4D6c6F5e6F7	Label displays 'Mimura Variation II'	Label displays 'Mimura Variation II'	Pass
			Played C4c3D3c5B4d2D6	Label displays 'Heath-Bat'	Label displays 'Heath-Bat'	Pass
			Played C4c3D3c5B4d2E2	Label displays 'Iwasaki Variation'	Label displays 'Iwasaki Variation'	Pass
			Played C4c3D3c5B4e3	Label displays 'Heath-Chimney'	Label displays 'Heath-Chimney'	Pass
			Played C4c3D3c5B5	Label displays 'Raccoon Dog'	Label displays 'Raccoon Dog'	Pass
			Played C4c3D3c5B6	Label displays 'Rocket'	Label displays 'Rocket'	Pass
			Played C4c3D3c5B6c6B5	Label displays 'Hamilton'	Label displays 'Hamilton'	Pass

			Played C4c3D3c5B6e3	Label displays 'Lollipop'	Label displays 'Lollipop'	Pass
			Played C4c3D3c5D6	Label displays 'Cow'	Label displays 'Cow'	Pass
			Played C4c3D3c5D6e3	Label displays 'Chimney'	Label displays 'Chimney'	Pass
			Played C4c3D3c5D6f4B4	Label displays 'Cow Bat/Bat/Cambridge'	Label displays 'Cow Bat/Bat/Cambridge'	Pass
			Played C4c3D3c5D6f4B4b6B5c6B3	Label displays 'Bat (Piau Continuation 2)'	Label displays 'Bat (Piau Continuation 2)'	Pass
			Played C4c3D3c5D6f4B4b6B5c6F5	Label displays 'Melnikov/Bat (Piau Continuation 1)'	Label displays 'Melnikov/Bat (Piau Continuation 1)'	Pass
			Played C4c3D3c5D6f4B4c6B5b3B6e3C2a4A5a6D2	Label displays 'Bat (Kling Continuation)'	Label displays 'Bat (Kling Continuation)'	Pass
			Played C4c3D3c5D6f4B4e3B3	Label displays 'Bat (Kling Alternative)'	Label displays 'Bat (Kling Alternative)'	Pass
			Played C4c3D3c5D6f4F5	Label displays 'Rose-v-Toth'	Label displays 'Rose-v-Toth'	Pass
			Played C4c3D3c5D6f4F5d2	Label displays 'Tanida'	Label displays 'Tanida'	Pass
			Played C4c3D3c5D6f4F5d2B5	Label displays 'Aircraft/Feldborg'	Label displays 'Aircraft/Feldborg'	Pass
			Played C4c3D3c5D6f4F5d2G4d7	Label displays 'Sailboat'	Label displays 'Sailboat'	Pass
			Played C4c3D3c5D6f4F5e6C6d7	Label displays 'Maruoka'	Label displays 'Maruoka'	Pass

			Played C4c3D3c5D6f4F5e6F6	Label displays 'Landau'	Label displays 'Landau'	Pass
			Played C4c3D3c5F6	Label displays 'Buffalo/Kenichi Variation'	Label displays 'Buffalo/Kenichi Variation'	Pass
			Played C4c3D3c5F6e2C6	Label displays 'Maruoka Buffalo'	Label displays 'Maruoka Buffalo'	Pass
			Played C4c3D3c5F6e3C6f5F4g5	Label displays 'Tanida Buffalo'	Label displays 'Tanida Buffalo'	Pass
			Played C4c3D3c5F6f5	Label displays 'Hokuriku Buffalo'	Label displays 'Hokuriku Buffalo'	Pass
			Played C4c3E6c5	Label displays 'Wing Variation'	Label displays 'Wing Variation'	Pass
			Played C4c3F5c5	Label displays 'Semi-Wing Variation'	Label displays 'Semi-Wing Variation'	Pass
			Played C4c5	Label displays 'Parallel Opening'	Label displays 'Parallel Opening'	Pass
			Played C4e3	Label displays 'Perpendicular Opening'	Label displays 'Perpendicular Opening'	Pass
			Played C4e3F4c5D6e6	Label displays 'Mimura'	Label displays 'Mimura'	Pass
			Played C4e3F4c5D6f3C6	Label displays 'Shaman/Danish'	Label displays 'Shaman/Danish'	Pass
			Played C4e3F4c5D6f3D3	Label displays 'Inoue'	Label displays 'Inoue'	Pass
			Played C4e3F4c5D6f3D3c3	Label displays 'Iago'	Label displays 'Iago'	Pass

			Played C4e3F4c5D6f3E2	Label displays 'Bhagat'	Label displays 'Bhagat'	Pass
			Played C4e3F4c5D6f3E6c3 D3e2	Label displays 'Rose'	Label displays 'Rose'	Pass
			Played C4e3F4c5D6f3E6c3 D3e2B5	Label displays 'Flat'	Label displays 'Flat'	Pass
			Played C4e3F4c5D6f3E6c3 D3e2B5f5	Label displays 'Rotating Flat'	Label displays 'Rotating Flat'	Pass
			Played C4e3F4c5D6f3E6c3 D3e2B5f5B3	Label displays 'Murakami Variation'	Label displays 'Murakami Variation'	Pass
			Played C4e3F4c5D6f3E6c3 D3e2B5f5B4f6C2e7 D2c7	Label displays 'Rotating Flat (Kling Continuation)'	Label displays 'Rotating Flat (Kling Continuation)'	Pass
			Played C4e3F4c5D6f3E6c3 D3e2B6f5	Label displays 'Rose-Birth'	Label displays 'Rose-Birth'	Pass
			Played C4e3F4c5D6f3E6c3 D3e2B6f5B4f6G5d7	Label displays 'Brightstein'	Label displays 'Brightstein'	Pass
			Played C4e3F4c5D6f3E6c3 D3e2B6f5G5	Label displays 'Rose-Birdie/Rose-Tamenori'	Label displays 'Rose-Birdie/Rose-Tamenori'	Pass
			Played C4e3F4c5D6f3E6c3 D3e2B6f5G5f6	Label displays 'Rose-Tamenori-Kling'	Label displays 'Rose-Tamenori-Kling'	Pass
			Played C4e3F4c5D6f3E6c3 D3e2D2	Label displays 'Greenberg/ Dawg'	Label displays 'Greenberg/ Dawg'	Pass
			Played C4e3F4c5D6f3E6c6	Label displays 'Ralle'	Label displays 'Ralle'	Pass
			Played C4e3F4c5E6	Label displays 'Horse'	Label displays 'Horse'	Pass

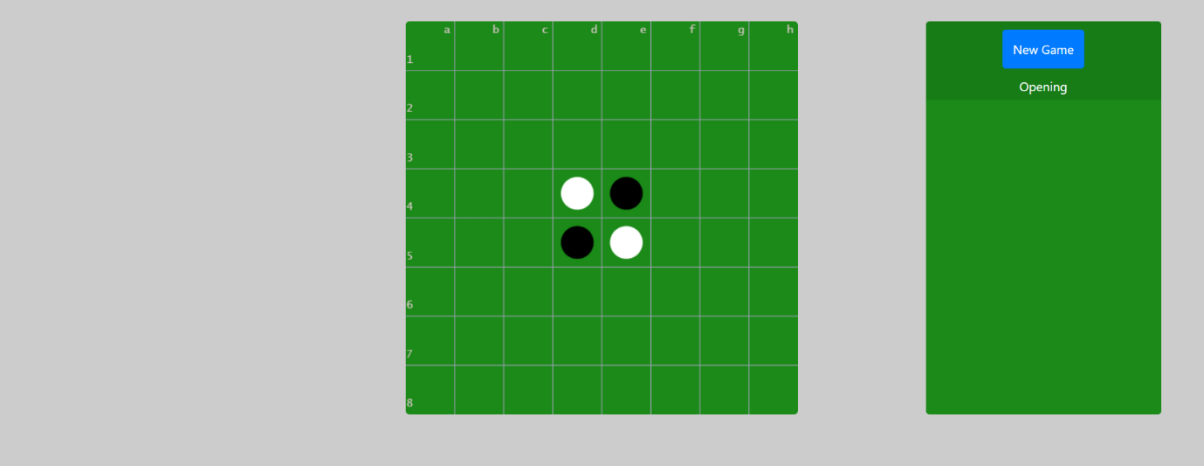
			Played C4e3F5b4	Label displays 'Ganglion/No-Cat'	Label displays 'Ganglion/No-Cat'	Pass
			Played C4e3F5b4F3	Label displays 'Swallow'	Label displays 'Swallow'	Pass
			Played C4e3F5b4F3f4E2e6G5f6D6c6	Label displays 'No-Cat (Continuation)'	Label displays 'No-Cat (Continuation)'	Pass
			Played C4e3F5e6D3	Label displays 'Italian'	Label displays 'Italian'	Pass
			Played C4e3F5e6F4	Label displays 'Cat'	Label displays 'Cat'	Pass
			Played C4e3F5e6F4c5D6c6F7f3	Label displays 'Sakaguchi'	Label displays 'Sakaguchi'	Pass
			Played C4e3F5e6F4c5D6c6F7g5G6	Label displays 'Berner'	Label displays 'Berner'	Pass
			Played C4e3F6b4	Label displays 'Bent Ganglion'	Label displays 'Bent Ganglion'	Pass
			Played C4e3F6e6F5	Label displays 'Tiger'	Label displays 'Tiger'	Pass
			Played C4e3F6e6F5c5C3	Label displays 'Stephenson'	Label displays 'Stephenson'	Pass
			Played C4e3F6e6F5c5C3b4	Label displays 'No-Kung'	Label displays 'No-Kung'	Pass
			Played C4e3F6e6F5c5C3b4D6c6B5a6B6c7	Label displays 'No-Kung (Continuation)'	Label displays 'No-Kung (Continuation)'	Pass
			Played C4e3F6e6F5c5C3c6	Label displays 'Comp'Oth'	Label displays 'Comp'Oth'	Pass
			Played C4e3F6e6F5c5C3c6D3d2E2b3C1c2B4a3A5b5A6a4A2	Label displays 'F.A.T. Draw'	Label displays 'F.A.T. Draw'	Pass

			Played C4e3F6e6F5c5C3c6D6	Label displays 'Lightning Bolt'	Label displays 'Lightning Bolt'	Pass
			Played C4e3F6e6F5c5C3g5	Label displays 'Kung'	Label displays 'Kung'	Pass
			Played C4e3F6e6F5c5D3	Label displays 'Leader's Tiger'	Label displays 'Leader's Tiger'	Pass
			Played C4e3F6e6F5c5D6	Label displays 'Brightwell'	Label displays 'Brightwell'	Pass
			Played C4e3F6e6F5c5F4g5G4f3C6d3D6	Label displays 'Ishii'	Label displays 'Ishii'	Pass
			Played C4e3F6e6F5c5F4g5G4f3C6d3D6b3C3b4E2b6	Label displays 'Mainline Tiger'	Label displays 'Mainline Tiger'	Pass
			Played C4e3F6e6F5c5F4g6F7	Label displays 'Rose-Bill'	Label displays 'Rose-Bill'	Pass
			Played C4e3F6e6F5c5F4g6F7d3	Label displays 'Tamenori'	Label displays 'Tamenori'	Pass
			Played C4e3F6e6F5c5F4g6F7g5	Label displays 'Central Rose-Bill/Dead Draw'	Label displays 'Central Rose-Bill/Dead Draw'	Pass
			Played C4e3F6e6F5g6	Label displays 'Aubrey/Tanaka'	Label displays 'Aubrey/Tanaka'	Pass
			Played C4e3F6e6F5g6E7c5	Label displays 'Aubrey (Feldborg Continuation)'	Label displays 'Aubrey (Feldborg Continuation)'	Pass
12	L12	Testing whether the most recently moved to square is highlighted	Played C4	C4 square is made darker	C4 square is made darker	Pass

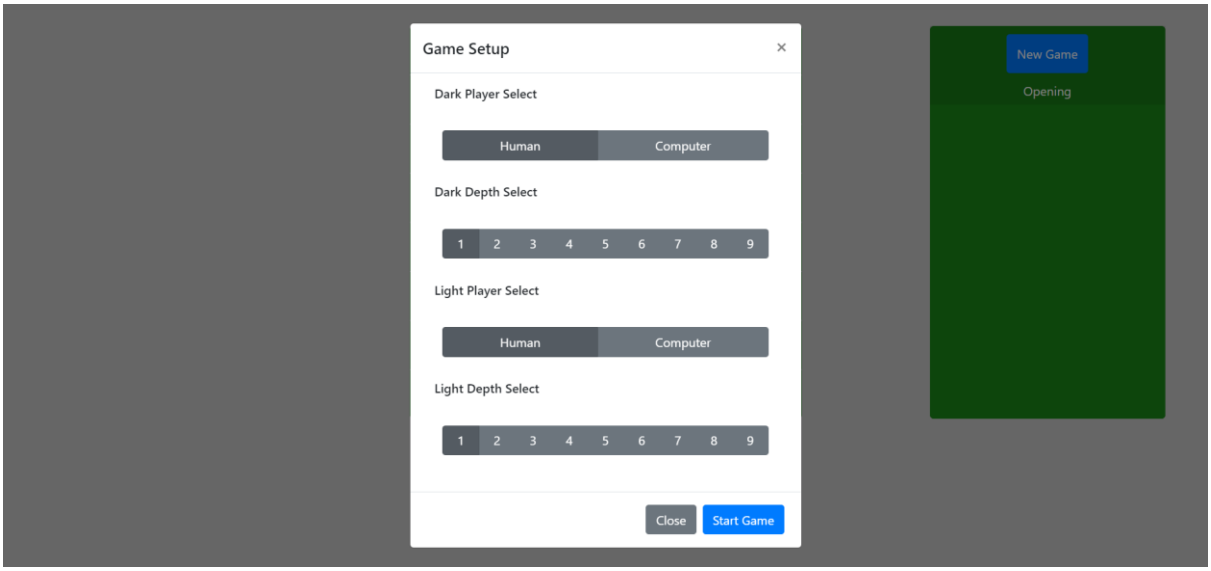
			Played C4c3	c3 square is made darker	c3 square is made darker	Pass
			Played C4c3D3c5B6e3	e3 square is made darker	e3 square is made darker	Pass
		Checking square highlighting works when a pass occurs	Played e6f6f5f4e3d3f3c5c2b1c3b2f7g7g6g8a1a2h8h6g5g2f2g1h1h2e2e1d2c1g4h3h5h4g3h7c4b4e7d8d6c7b5b6c6b7d7c8b3a4a6a5d1a7f1--	f1 square is made darker	f1 square is made darker	Pass
13	L13	Checking whether, after a game has finished, a window pops up displaying the score and winner	Played f5d6c5f4d3e3g4g3f3e2f2e1e6e7f6c6d7c7d8e8f8d2c4b3b4c3c8h4d1f1g5c1h5c2a3a5a4b5a6h6g6f7h3b2a1g2b6h2h1a7a8g1b7b8g7b1h7h8g8a2	A window pops up and "Dark Won 39-25" is displayed	A window pops up and "Dark Won 39-25" is displayed	Pass
14	L14a,b,c	Checking whether, in the post game popup window, a table appears displaying, for each move, the desirability of the resultant position, a qualitative description of the move, and a suggested optimal move.	Played d3c5f6f5e6e3d6e7d7c6f3c4f4c2c3d2b5f2b4b3c7a4d8a5b6a6f1c8e2e1a3g6f7e8f8g5g4h3h4g3h5h6g7g1c1a2d1b1b2h7b8a1g8b7h2h1g2a7a8h8	A table appears in the post game window displaying, for each move: The numical desirability of the move, a qualitative description of the move, and a suggested optimal move	A table appears in the post game window displaying, for each move: The numical desirability of the move, a qualitative description of the move, and a suggested optimal move	Pass

Evidence for Tests 1 -14

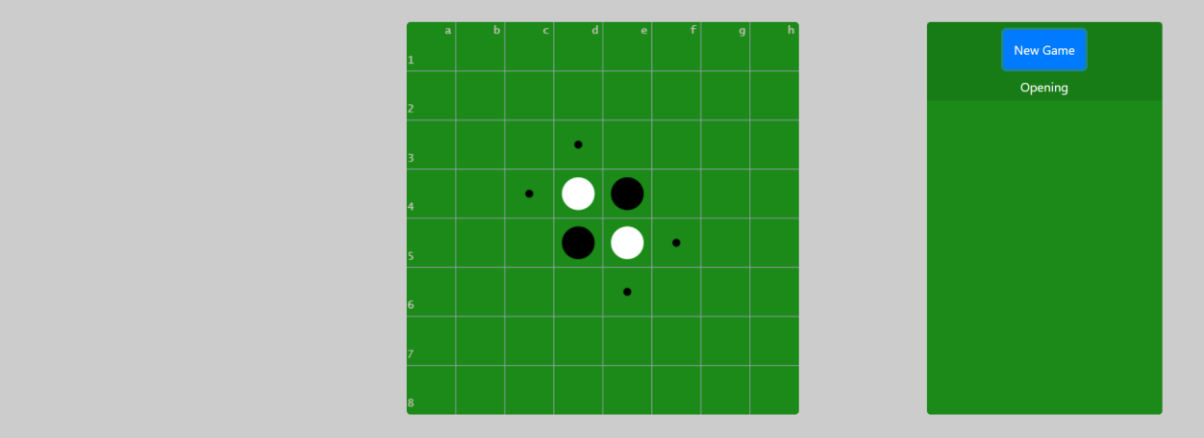
Test 1


a b c d e f g h
1
2
3
4
5
6
7
8
New Game
Opening

Test 2


Game Setup
Dark Player Select
Human
Computer
Dark Depth Select
1 2 3 4 5 6 7 8 9
Light Player Select
Human
Computer
Light Depth Select
1 2 3 4 5 6 7 8 9
Close
Start Game
New Game
Opening

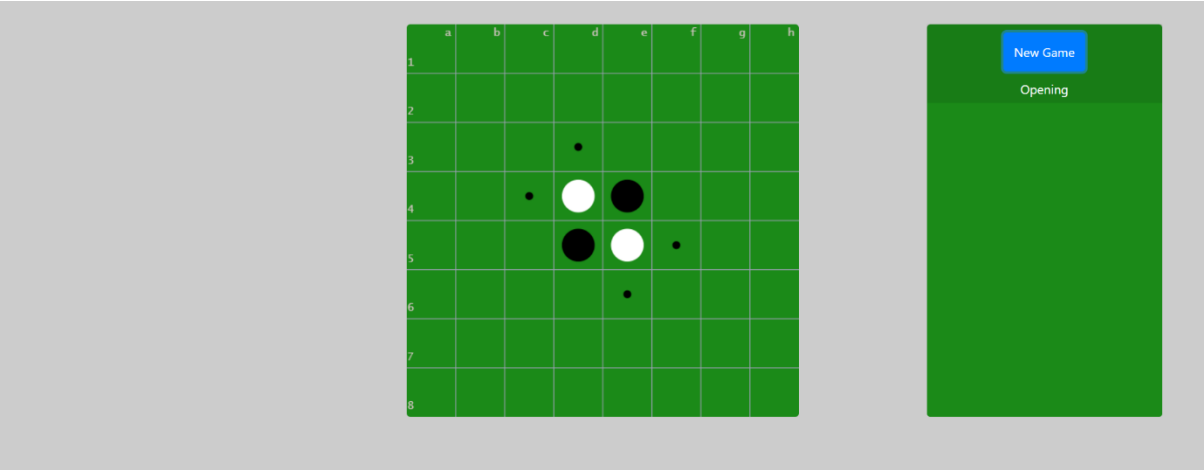
Test 3



Test 4

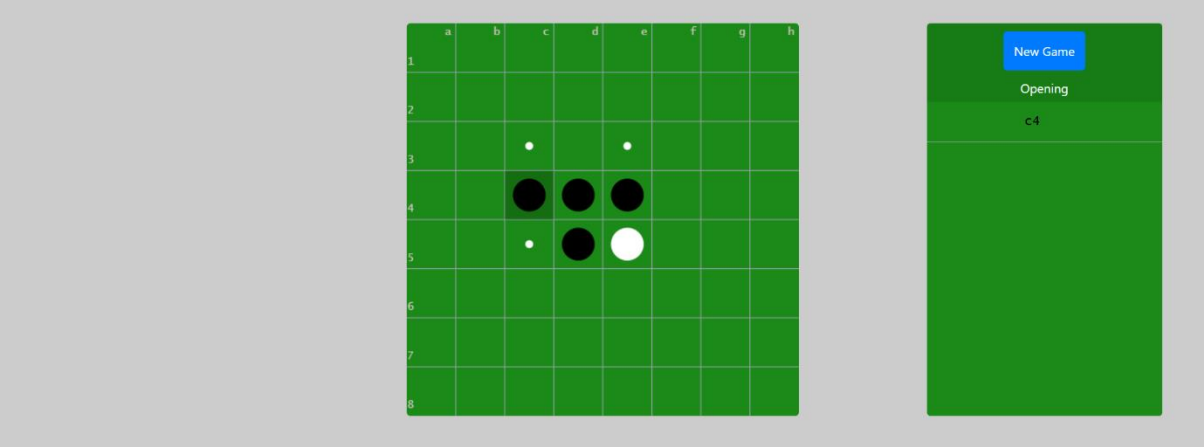
This test was carried out through the console. However, the evidence screenshots for test 5 also demonstrate the accuracy of the legal move finding function.

Test 5

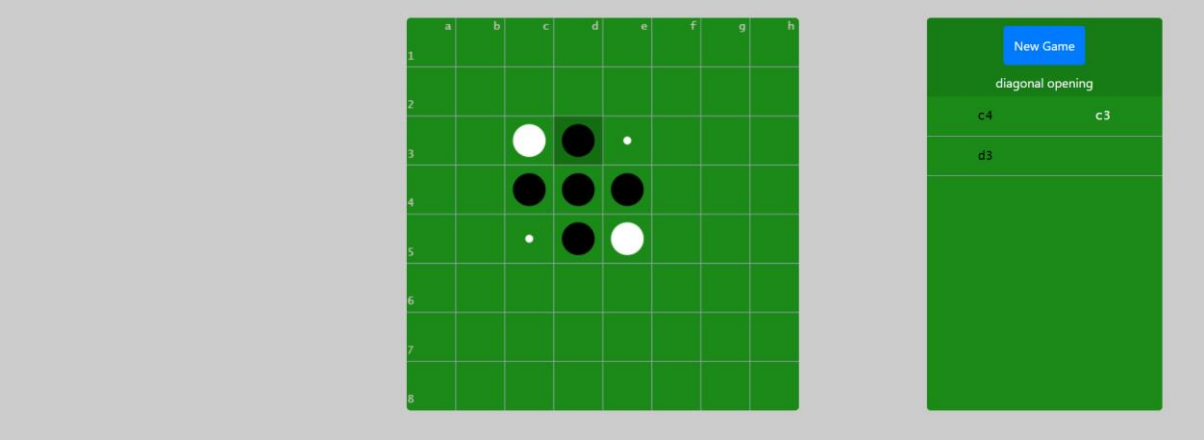


Dark's available moves are displayed as black dots.

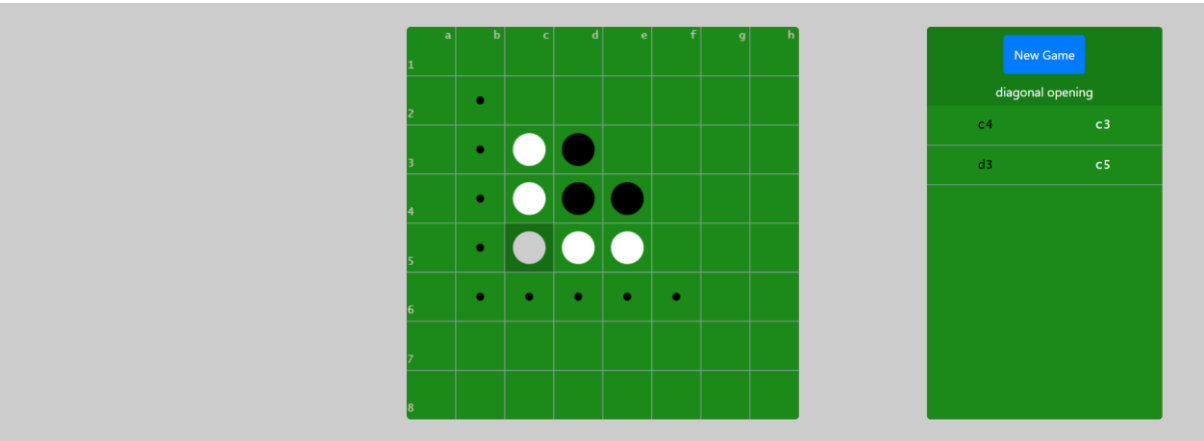
Light's available moves are displayed as white dots.



Test 6



After clicking in the c5 square, a white token is inserted at c5 and the display as per the next screenshot.

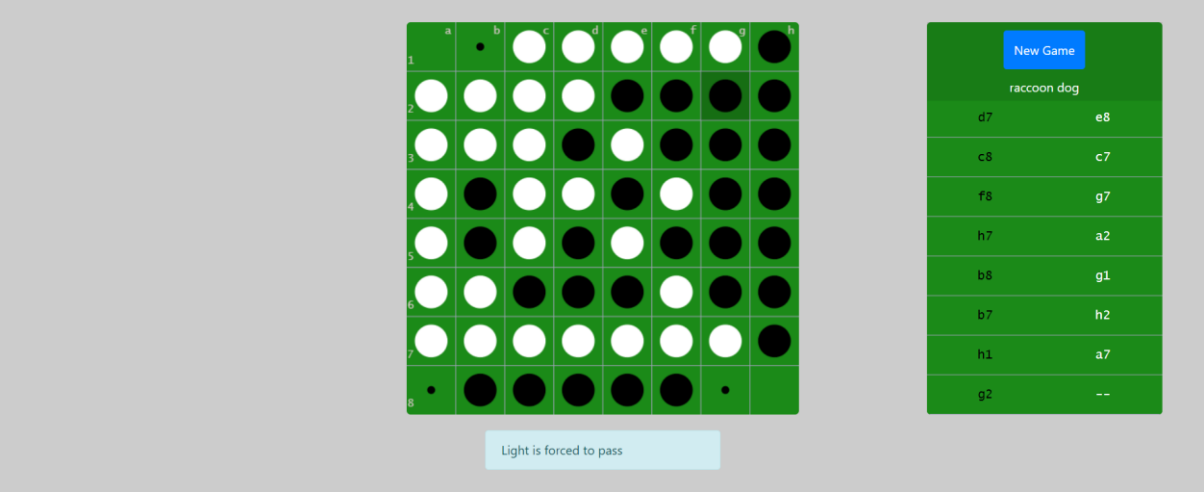


This screenshot shows the board having flipped the relevant discs as this is done immediately upon a move being inputted and it is not possible in the final version of the application to display the board in between inserting a token and flipping discs.

Test 7

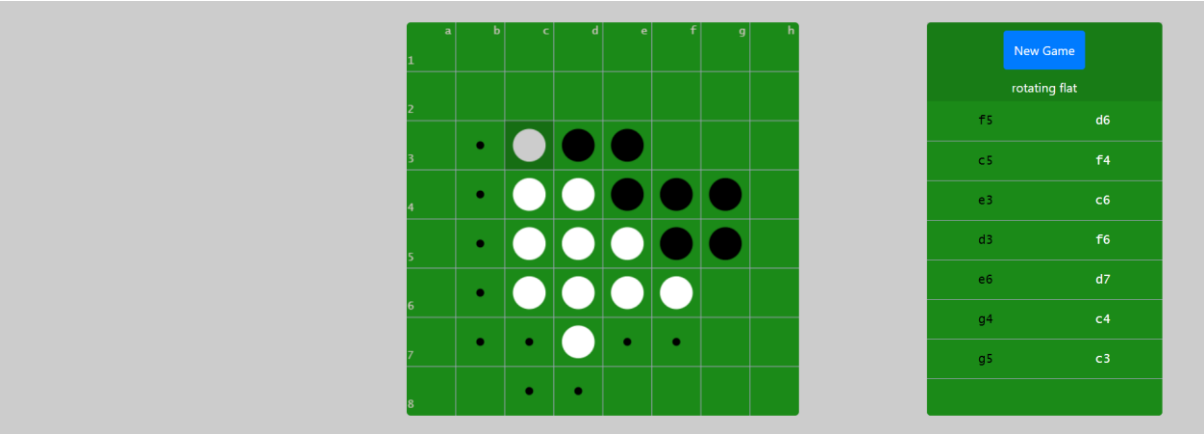
It is not possible to provide evidence for this test in the form of screenshots.

Test 8

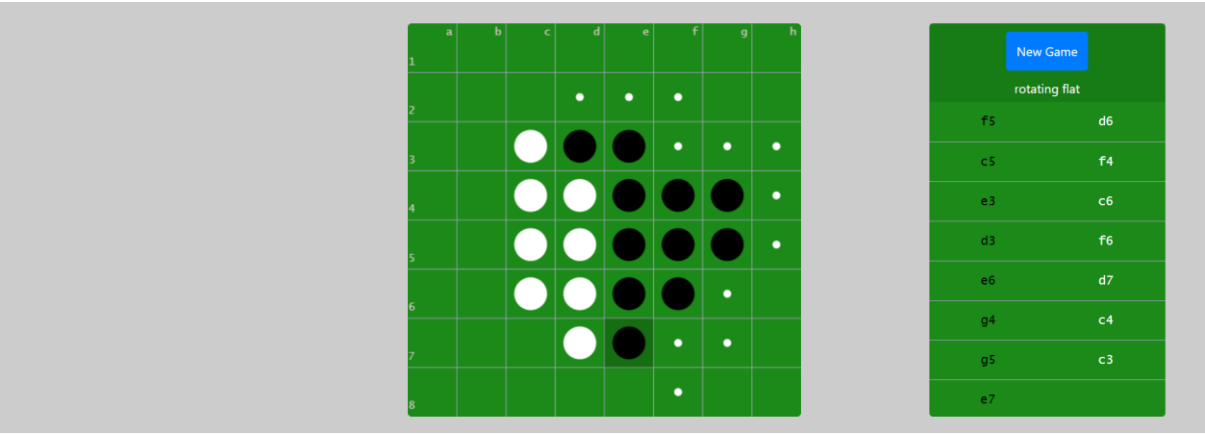


Test 9

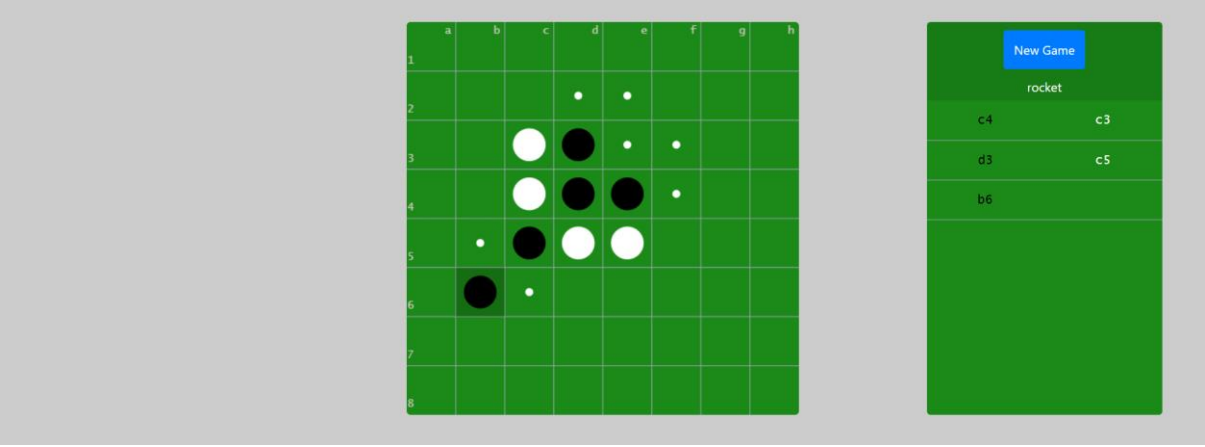
Testing for dark:



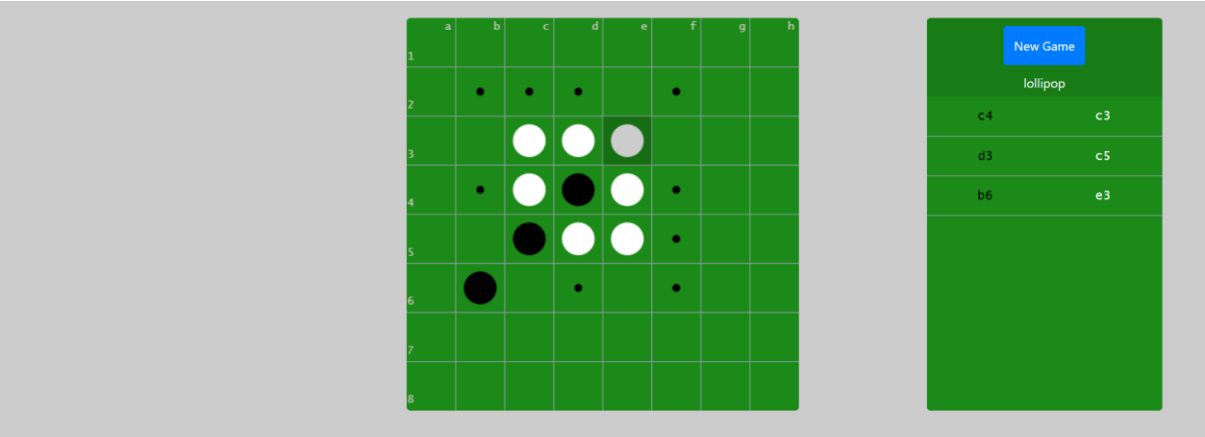
Dark plays e7 resulting in e5, e6, and f6 being flipped, as is shown in the screenshot below.



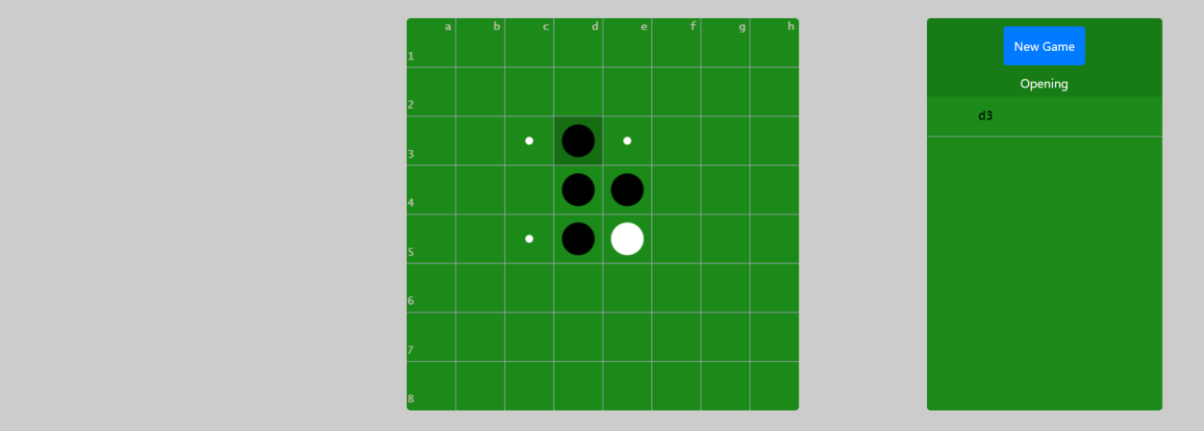
Testing for light:



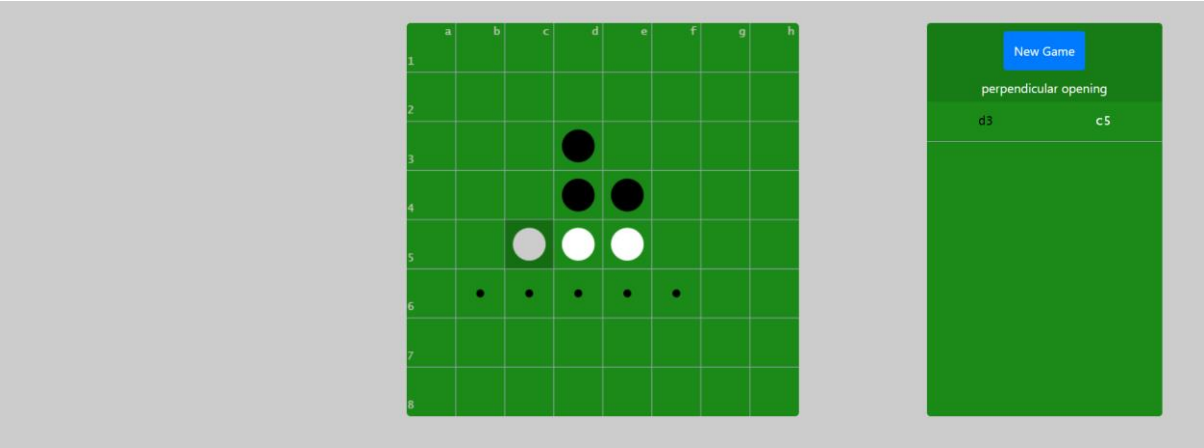
Light plays e3 resulting in d3 and e4 being flipped, as shown in the screenshot below.



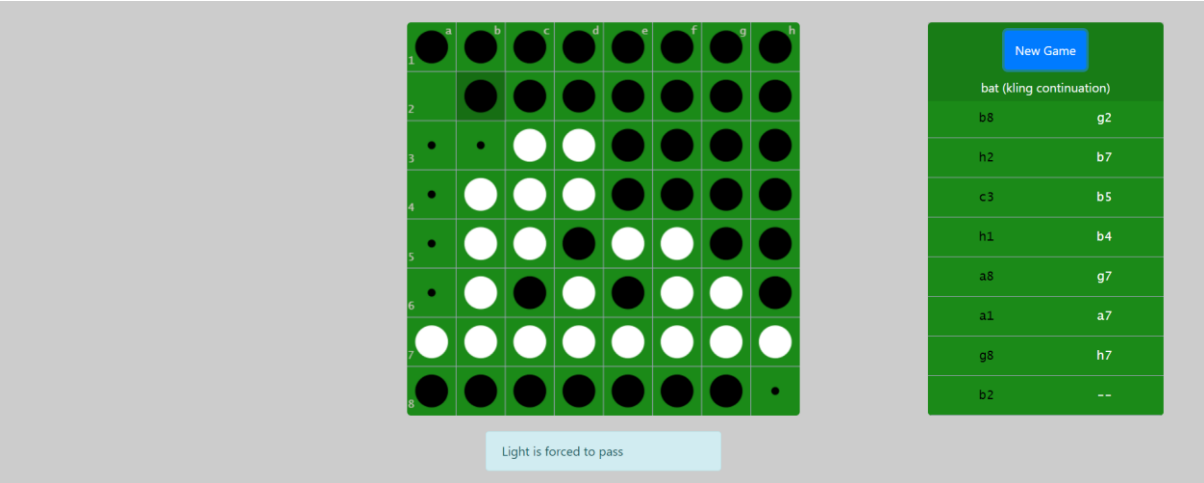
Test 10



In the above screenshot, dark plays d3 and, as expected, the game table on the right of the screen is updated to show d3 in the left hand column.



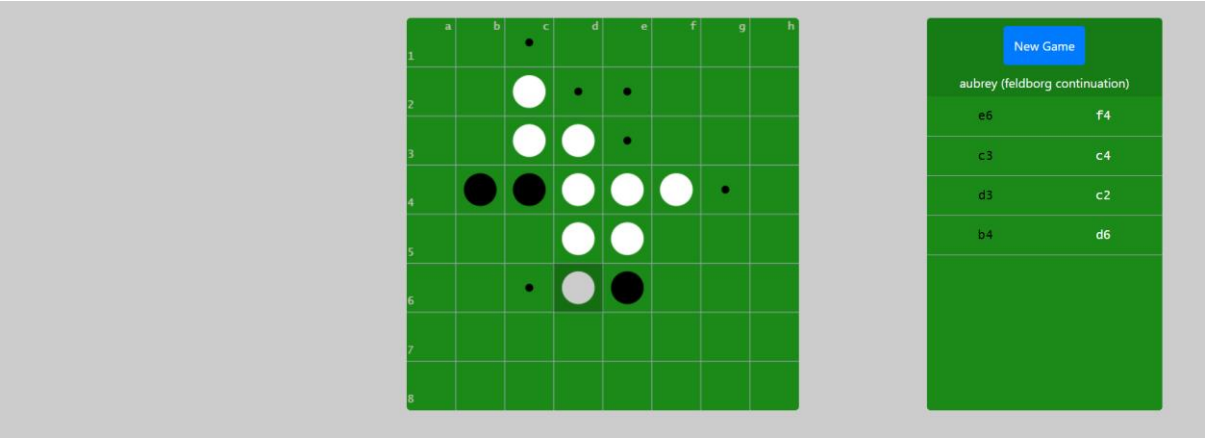
In the above screenshot, light plays c5 and, as expected, the game table on the right of the screen is updated to show c5 in the right hand column.



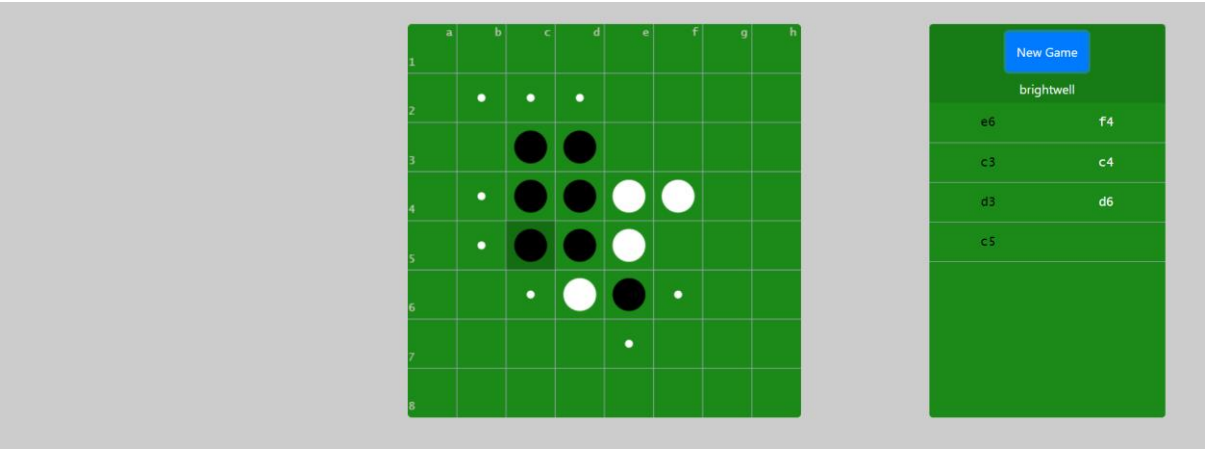
In the above screenshot, light is forced to pass and, as expected "- -" is displayed in the game table in the right hand column.

Test 11

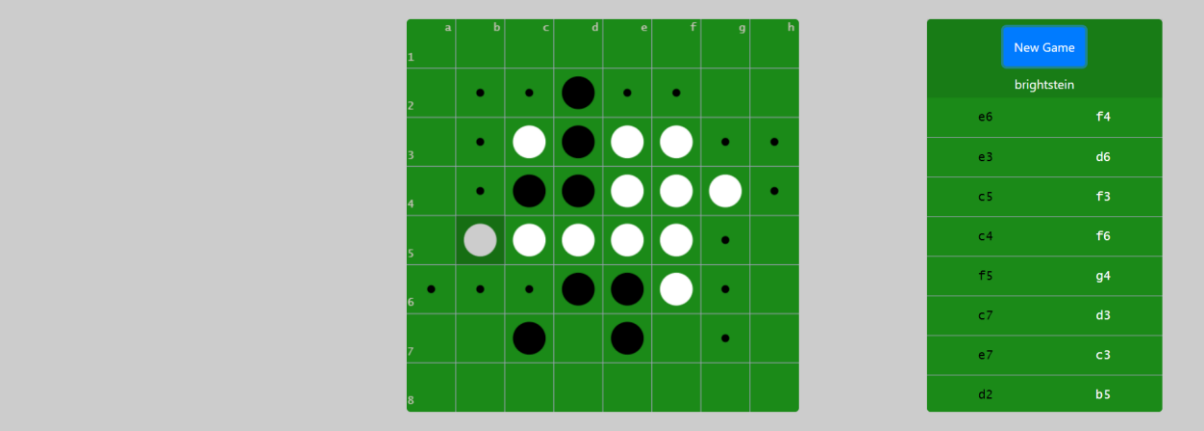
In order to demonstrate the process of testing whether the application correctly displays the current opening in the right hand panel, I have presented the screenshots below which represent a sample of the different openings that were tested. For reference, a transcript of the opening is also given to confirm that the opening have been labelled correctly. In most cases, the transcript does not match the game played out in the screenshot. This is because, for each opening, there are four equivalent ways in which it can be played, due to the fourfold symmetry of the starting position.



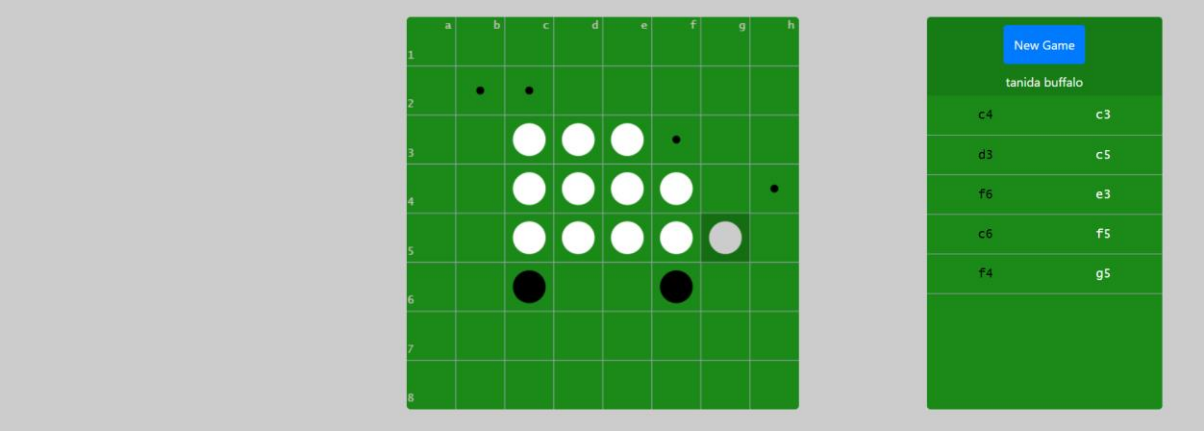
Aubrey (Feldborg Continuation): C4e3F6e6F5g6E7c5



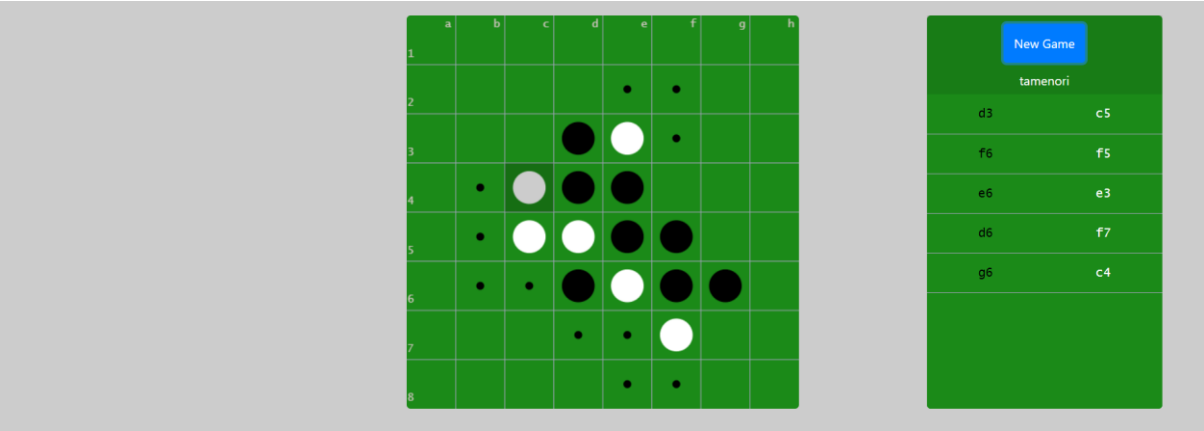
Brightwell: C4e3F6e6F5c5D6



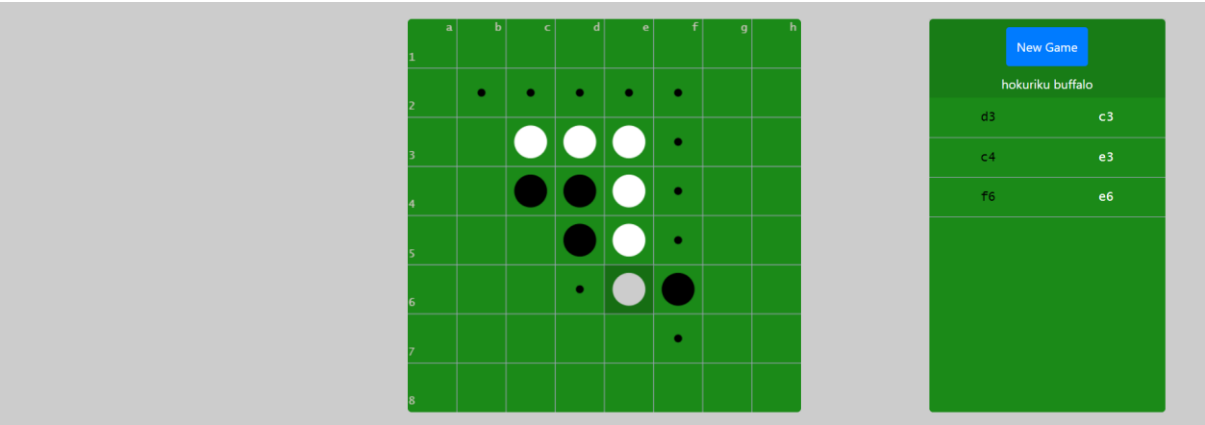
Brightstein: C4e3F4c5D6f3E6c3D3e2B6f5B4f6G5d7



Tanida Buffalo: C4c3D3c5F6e3C6f5F4g5



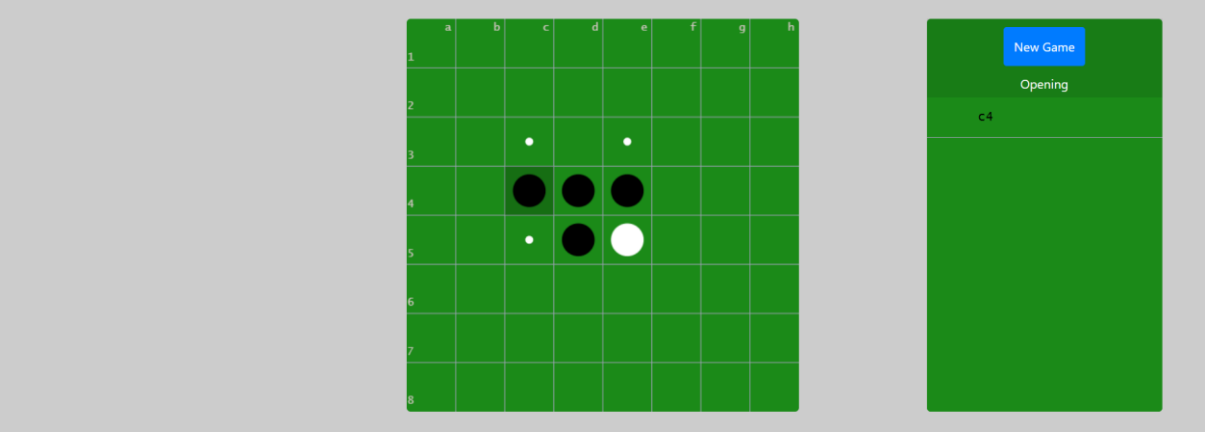
Tamenori: C4e3F6e6F5c5F4g6F7d3



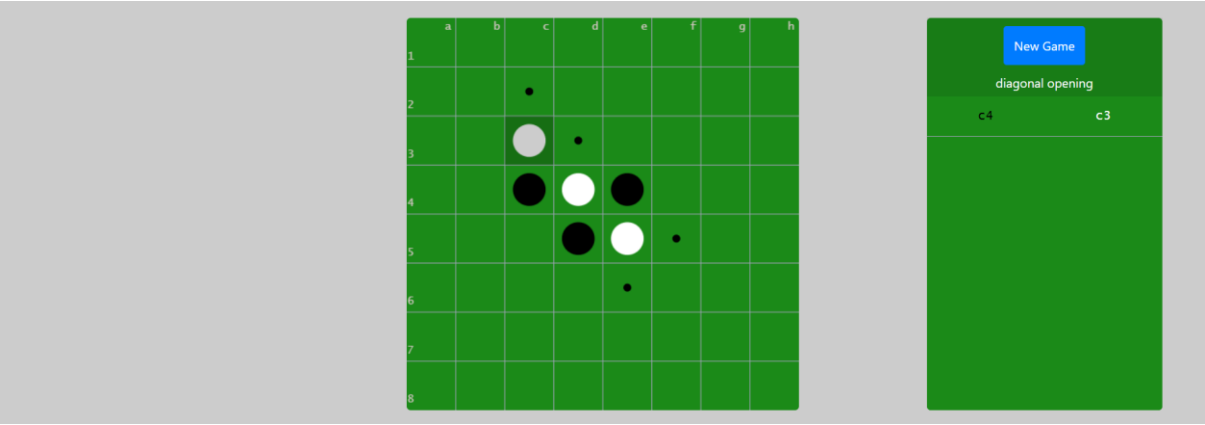
Hokuriku Buffalo: C4c3D3c5F6f5

Test 12

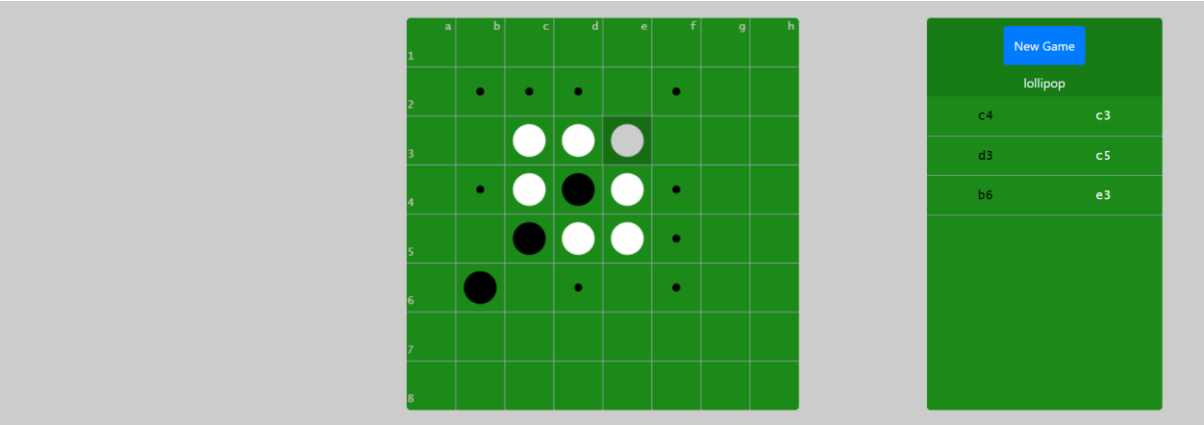
As shown in the screenshot below, the c4 square is made darker to indicate that c4 was the last move played.



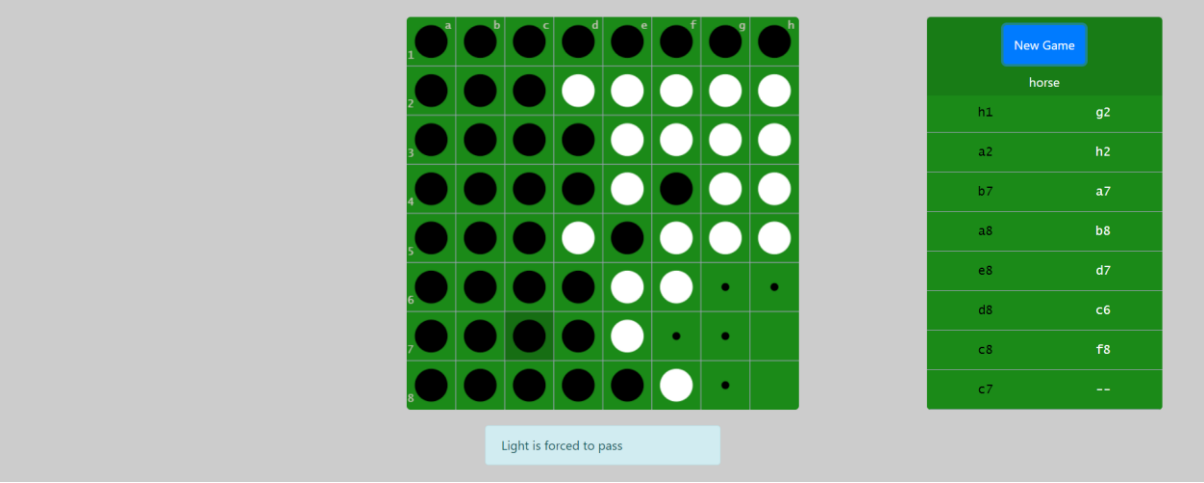
Similarly, in the screenshot below, the c3 square is made darker as expected.



Here, the e3 square is highlighted as expected.

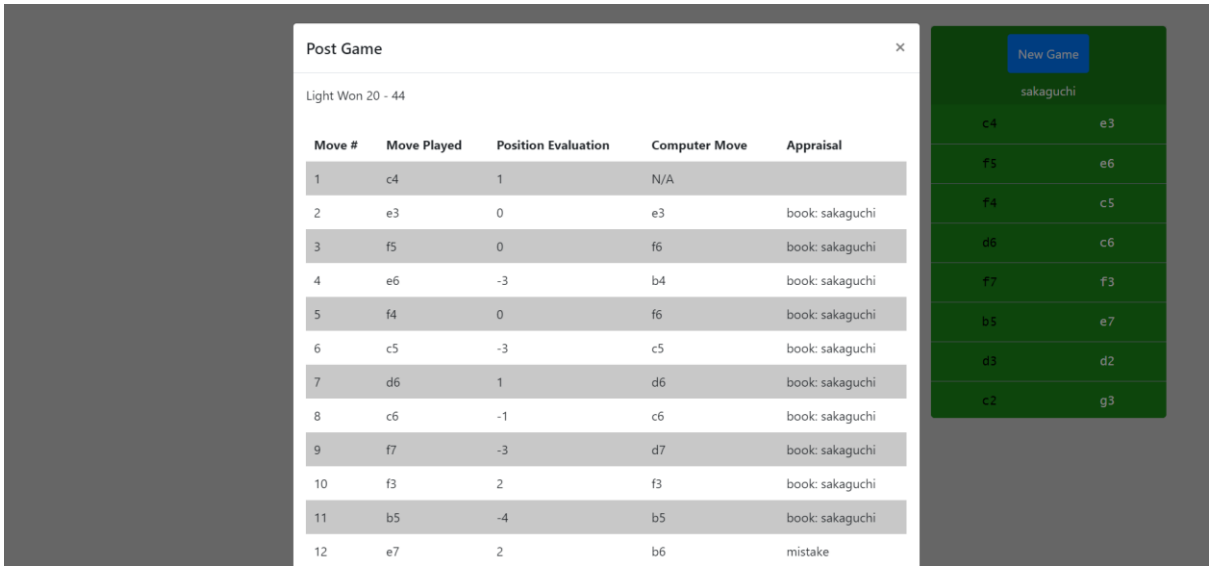


The screenshot below demonstrates that, when a played is forced to pass, the last square to be played into is still highlighted (c7 in this case).



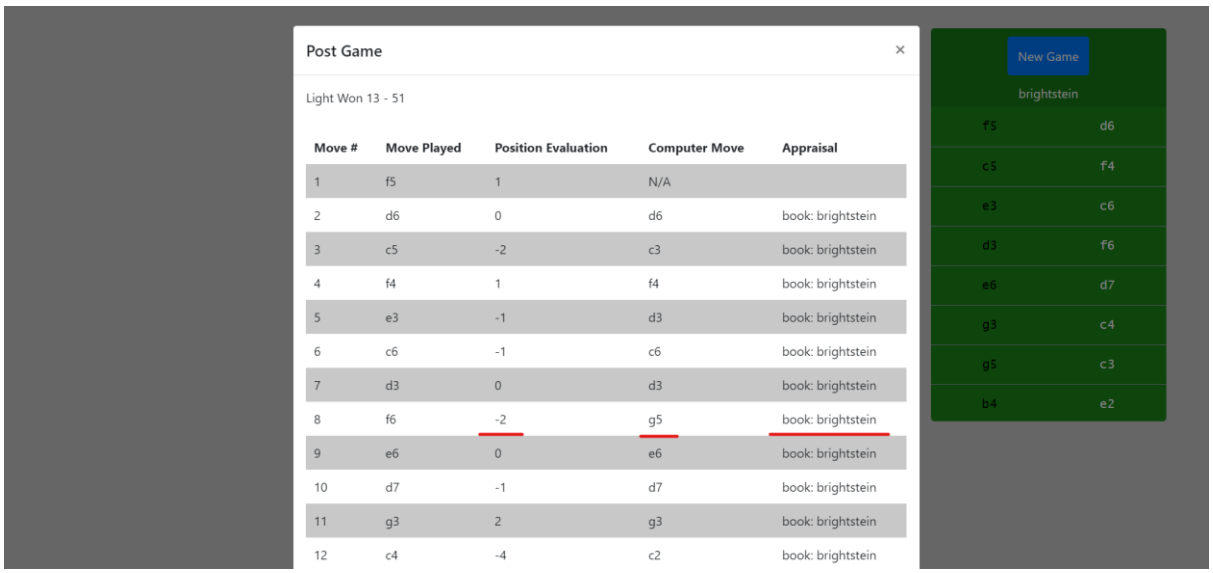
Test 13

The screenshot below demonstrates that after a game ends, a popup window opens displaying the result of the game. In this instance, the game played was: c4e3f5e6f4c5d6c6f7f3b5e7d3d2c2g3e8c3g4h4h3h5h6c1c7f8b4b3d1d8e2a3a5a4a6c8d7b6f2a7b7f6b2a1g6g7b1g5h8a2h7e1f1h2g2b8h1g1a8g8. As expected, the application reports that light won by 20-44.

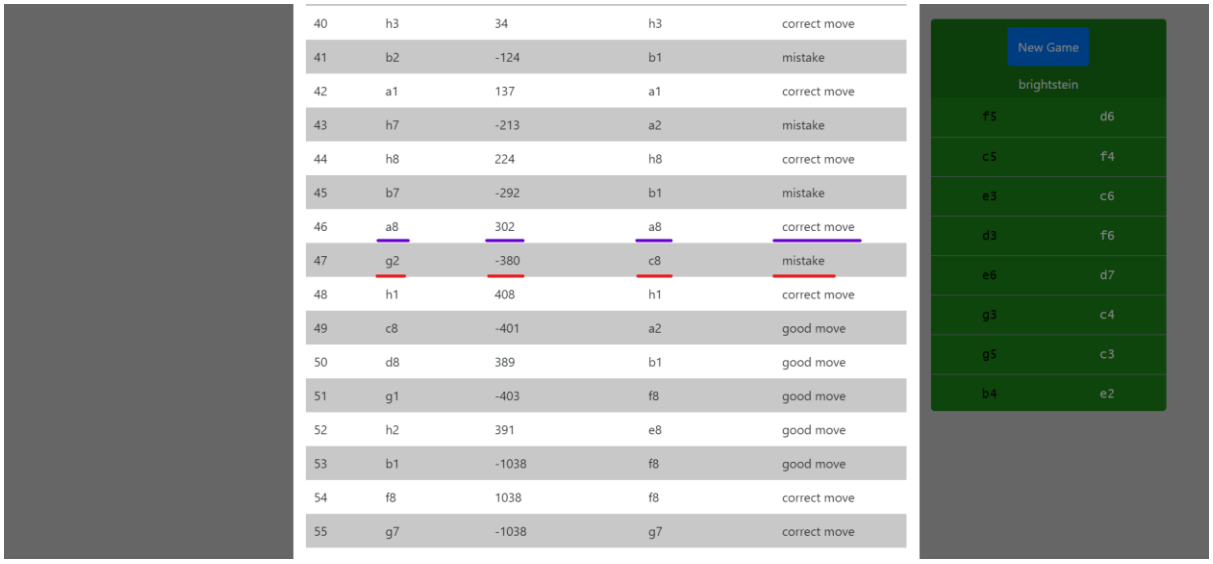


Test 14

The annotated screenshot below demonstrates that, for each move, the application's post-game table displays: a position evaluation (how desirable that position is for the player whose turn it was); the move that the computer on search depth 3 would have played, and a qualitative appraisal of the move. In this screenshot, the move that is annotated, f6, is simply a book move and so the appraisal column displays the opening that is being played.



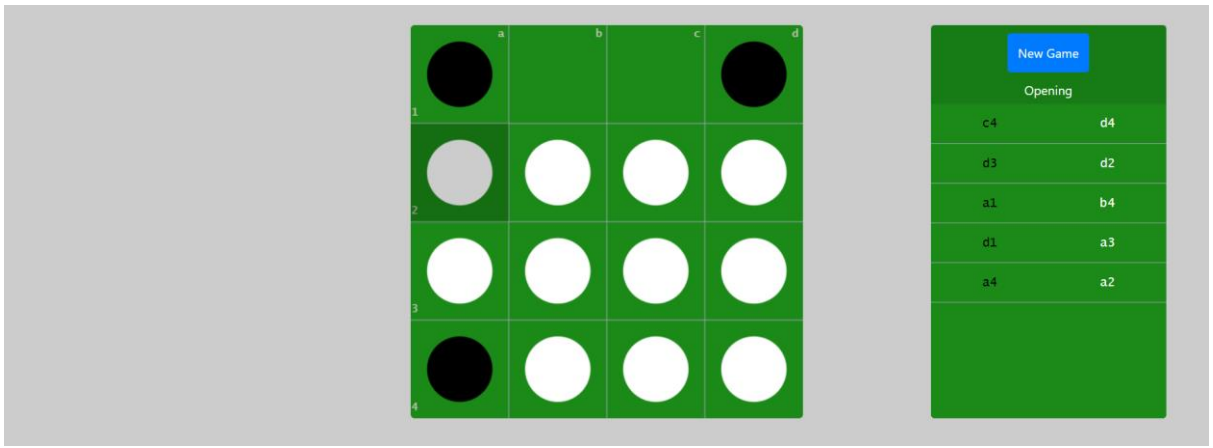
The screenshot below is a continuation of the game report above and demonstrates some of the different qualitative labels that application assigns to moves. Move 46, which is highlighted in purple is labelled as a 'correct move' since it is the same as the computer's suggested move. By contrast, move 47, highlighted in red, is deemed a mistake since it is not the 'correct move' and, in addition, dark has increased their disadvantage compared to the last move (-380 < -302). Moves which are not 'correct' but nevertheless increase the value of the position evaluation are labelled 'good' moves, as is the case with move 49.



40	h3	34	h3	correct move
41	b2	-124	b1	mistake
42	a1	137	a1	correct move
43	h7	-213	a2	mistake
44	h8	224	h8	correct move
45	b7	-292	b1	mistake
46	a8	302	a8	correct move
47	g2	-380	c8	mistake
48	h1	408	h1	correct move
49	c8	-401	a2	good move
50	d8	389	b1	good move
51	g1	-403	f8	good move
52	h2	391	e8	good move
53	b1	-1038	f8	good move
54	f8	1038	f8	correct move
55	g7	-1038	g7	correct move

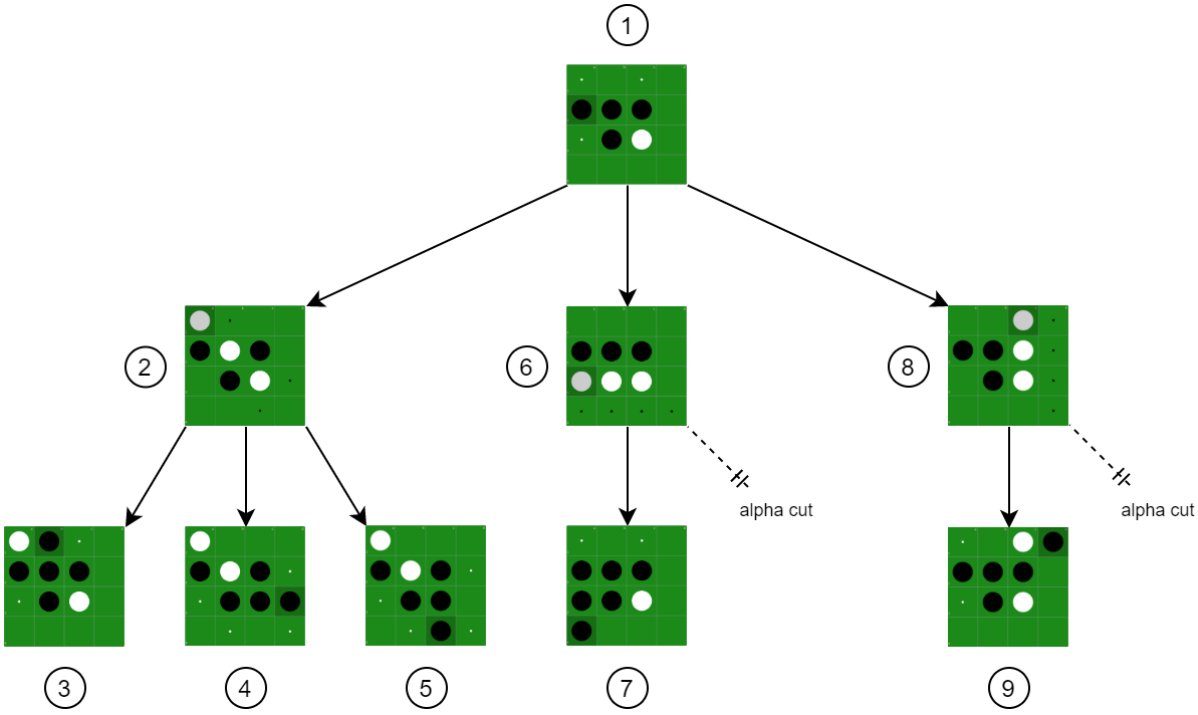
Test 15: Testing the Alphabeta Search

Since, in most cases, it will be infeasible to manually compute the correct output of an alpha-beta search performed on an 8x8 Reversi position, a 4x4 board will be used to test the alpha-beta search algorithm. Reversi is 'solved' for a 4x4 board in the sense that, if both players play perfectly, it is known that light wins 11-3 Therefore, an easy to perform but nevertheless instructive test of whether the alpha-beta search is working correctly would be to see if this result is achieved on a 4x4 board by two computer players set to a search depth of 12 or more (since 12 is the maximum number of moves in a 4x4 game). The result of this test is shown in the screenshot below.



As shown in the screenshot, the result of two computer players at search depth 12 playing each other on a 4x4 board is an 11-3 win for white as expected. The moves played are displayed in the game table on the right-hand side of the screen. Note that it is not necessary to validate whether this sequence of moves matches existing solutions for the 4x4 game as, clearly, all solutions which result in a 11-3 win for white are equivalent. Therefore, the application passes this test, giving a high degree of confidence that the alpha-beta search is functioning correctly.

Although the fact that the computer plays the 4x4 game perfectly does suggest that the alpha-beta search is functioning correctly, in order to make the testing more rigorous, I will manually perform an alpha-beta search on a very shallow game tree and compare my working with the console logs of the application performing the same search. I will search to a depth of two as this allows for pruning to occur and, since the choice of the first move is inconsequential, I will start from a position following dark's first move. To make commentary of the process easier to follow, the resulting game tree is presented below, with the pruned branches omitted to save space.



When manually computing the alpha-beta search, I constructed the trace table below, which includes commentary where relevant. The rows highlighted in green indicate that the value of that node has been calculated for the final time and, therefore, its assigned value at that point represents its true value.

Current Node	Maximising Player	Depth	Alpha	Beta	Value	Most Promising Node	Comment
1	TRUE	2	-10000	10000			
2	FALSE	1	-10000	10000			
3	TRUE	0	-10000	10000	142		Evaluated at 142
2	FALSE	1	-10000	142	142	3	Value updated Beta updated
4	TRUE	0	-10000	142	146		Evaluated at 146
2	FALSE	1	-10000	142	142	3	142 < 146 so neither Value nor Beta are updated
5	TRUE	0	-10000	142	146		Evaluated at 146
2	FALSE	1	-10000	142	142	3	142 < 146 so neither Value nor Beta are updated
1	TRUE	2	142	10000	142	2	Value updated Alpha Updated
6	FALSE	1	142	10000			
7	TRUE	0	142	10000	-57		Evaluated at -57
6	FALSE	1	142	-57	-57	7	Value updated Beta updated
Alpha > Beta => Alpha cut occurs							
6	FALSE	1	142	-57	-57	7	
1	TRUE	2	142	10000	142	2	Value and Alpha remain unchanged
8	FALSE	1	142	10000			
9	TRUE	0	142	10000	-79		Evaluated at -79
8	FALSE	1	142	-79	-79	9	Value updated Beta updated
Alpha > Beta => Alpha cut occurs							
8	FALSE	1	142	-79	-79	9	
1	TRUE	2	142	10000	142	2	Value and Alpha remain unchanged

As shown in the trace table, the most promising node from node 1 is node 2 at the point at which the algorithm finishes. As such, white should play a1 in this position. In order to test whether the application is performing this process correctly, the alphabeta function will be modified to generate console logs at points which correspond as closely as possible to the rows of the above trace table. The resulting console logs are provided below.

```
alphabeta call at position:
[0, 0, 0, 0]
[1, 1, 1, 0]
[0, 1, 2, 0]
[0, 0, 0, 0]
maximising player: true | depth: 2 | alpha: -10000 | beta: 10000

alphabeta call at position:
[2, 0, 0, 0]
[1, 2, 1, 0]
[0, 1, 2, 0]
[0, 0, 0, 0]
maximising player: false | depth: 1 | alpha: -10000 | beta: 10000

alphabeta call at position:
[2, 1, 0, 0]
[1, 1, 1, 0]
[0, 1, 2, 0]
[0, 0, 0, 0]
maximising player: true | depth: 0 | alpha: -10000 | beta: 10000
leaf position evaluated at: 142

for position:
[2, 0, 0, 0]
[1, 2, 1, 0]
[0, 1, 2, 0]
[0, 0, 0, 0]
where maximising player = false | depth = 1
value and associated move: 142  [1, 0]
alpha: -10000 | beta: 142

alphabeta call at position:
[2, 0, 0, 0]
[1, 2, 1, 0]
[0, 1, 1, 1]
[0, 0, 0, 0]
maximising player: true | depth: 0 | alpha: -10000 | beta: 142
leaf position evaluated at: 146

for position:
[2, 0, 0, 0]
[1, 2, 1, 0]
[0, 1, 2, 0]
[0, 0, 0, 0]
where maximising player = false | depth = 1
value and associated move: 142  [1, 0]
alpha: -10000 | beta: 142
```

```
alphabeta call at position:
[2, 0, 0, 0]
[1, 2, 1, 0]
[0, 1, 1, 0]
[0, 0, 1, 0]
maximising player: true | depth: 0 | alpha: -10000 | beta: 142
leaf position evaluated at: 146

for position:
[2, 0, 0, 0]
[1, 2, 1, 0]
[0, 1, 2, 0]
[0, 0, 0, 0]
where maximising player = false | depth = 1
value and associated move: 142  [1, 0]
alpha: -10000 | beta: 142

position:
[2, 0, 0, 0]
[1, 2, 1, 0]
[0, 1, 2, 0]
[0, 0, 0, 0]
where maximising player = false | depth = 1 | alpha = -10000 | beta = 142
assigned a final value of: 142
with best move:   [1, 0]

for position:
[0, 0, 0, 0]
[1, 1, 1, 0]
[0, 1, 2, 0]
[0, 0, 0, 0]
where maximising player = true | depth = 2
value and associated move: 142  [0, 0]
alpha: 142 | beta: 10000

alphabeta call at position:
[0, 0, 0, 0]
[1, 1, 1, 0]
[2, 2, 2, 0]
[0, 0, 0, 0]
maximising player: false | depth: 1 | alpha: 142 | beta: 10000

alphabeta call at position:
[0, 0, 0, 0]
[1, 1, 1, 0]
[1, 1, 2, 0]
[1, 0, 0, 0]
maximising player: true | depth: 0 | alpha: 142 | beta: 10000
```

```
leaf position evaluated at: -57

for position:
[0, 0, 0, 0]
[1, 1, 1, 0]
[2, 2, 2, 0]
[0, 0, 0, 0]
where maximising player = false | depth = 1
value and associated move: -57  [0, 3]
alpha: 142 | beta: -57

alpha cut

position:
[0, 0, 0, 0]
[1, 1, 1, 0]
[2, 2, 2, 0]
[0, 0, 0, 0]
where maximising player = false | depth = 1 | alpha = 142 | beta = -57
assigned a final value of: -57
with best move:   [0, 3]

for position:
[0, 0, 0, 0]
[1, 1, 1, 0]
[0, 1, 2, 0]
[0, 0, 0, 0]
where maximising player = true | depth = 2
value and associated move: 142  [0, 0]
alpha: 142 | beta: 10000

alphabeta call at position:
[0, 0, 2, 0]
[1, 1, 2, 0]
[0, 1, 2, 0]
[0, 0, 0, 0]
maximising player: false | depth: 1 | alpha: 142 | beta: 10000

alphabeta call at position:
[0, 0, 2, 1]
[1, 1, 1, 0]
[0, 1, 2, 0]
[0, 0, 0, 0]
maximising player: true | depth: 0 | alpha: 142 | beta: 10000
leaf position evaluated at: -79

for position:
[0, 0, 2, 0]
```

```
[1, 1, 2, 0]
[0, 1, 2, 0]
[0, 0, 0, 0]
where maximising player = false | depth = 1
value and associated move: -79  [3, 0]
alpha: 142 | beta: -79

alpha cut

position:
[0, 0, 2, 0]
[1, 1, 2, 0]
[0, 1, 2, 0]
[0, 0, 0, 0]
where maximising player = false | depth = 1 | alpha = 142 | beta = -79
assigned a final value of: -79
with best move:   [3, 0]

for position:
[0, 0, 0, 0]
[1, 1, 1, 0]
[0, 1, 2, 0]
[0, 0, 0, 0]
where maximising player = true | depth = 2
value and associated move: 142  [0, 0]
alpha: 142 | beta: 10000

position:
[0, 0, 0, 0]
[1, 1, 1, 0]
[0, 1, 2, 0]
[0, 0, 0, 0]
assigned a final value of: 142
with best move:   [0, 0]
```

As shown by the console logs, the application also reaches the conclusion that light should play a1, since [0,0] is the index corresponding to the a1 square. Furthermore, on examination, it can be seen that these console logs are equivalent to the manually constructed trace table. Not only are the nodes visited in the same order, but the values of the variables match up for any given point in the execution of the algorithm. As such, this test is passed and we can conclude that the alpha-beta search is being performed correctly.

Evaluation

In order to evaluate the overall effectiveness of the solution, I will revisit the high-level objectives set out in the Analysis section and, for each objective, assess the extent to which it has been achieved.

H1. The user should be able to select the difficulty of the computer opponent.

The user can select the depth to which the computer player searches during the early and midgame which is equivalent to selecting the difficulty since a deeper alpha-beta search will result in the computer playing better. Regardless of what depth is selected, the computer player will play perfectly when there are 10 or fewer empty spaces on the board. This ensures that the computer does not 'throw' the endgame and, since the game tree's branching factor decreases in the endgame, searching to a depth of 10 is nowhere near as computationally expensive in the endgame as it is during the midgame. This feature works well in the sense that, when set to a higher search depth, the application does not play disproportionately badly in the endgame. However, on the shallowest depth settings, this feature often results in the computer playing the endgame too well and denying beginners a chance to defeat the computer player on its easiest settings. One possible future enhancement, therefore, would be to make the number of moves played perfectly at the end dependant on the search depth selected for the midgame.

In summary, the user can select the depth to which the computer player searches in the midgame and so this first objective has been achieved. However, the application could be improved by the addition of more sophisticated depth customisation options, such as the ability to determine the number of moves from the end from which the computer player plays perfectly.

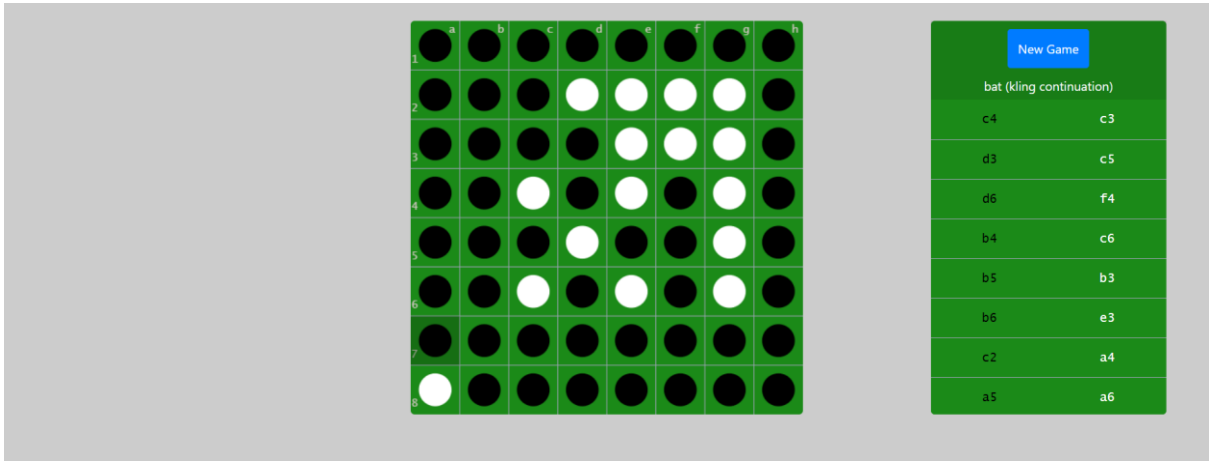
H2. The program should have a post-game analysis feature which shows how well the player played throughout the game, allowing the user to identify which phases of the game should be improved.

H3. The application should be able to identify blunders and suggest better moves. This functionality should be available in the post-game analysis feature.

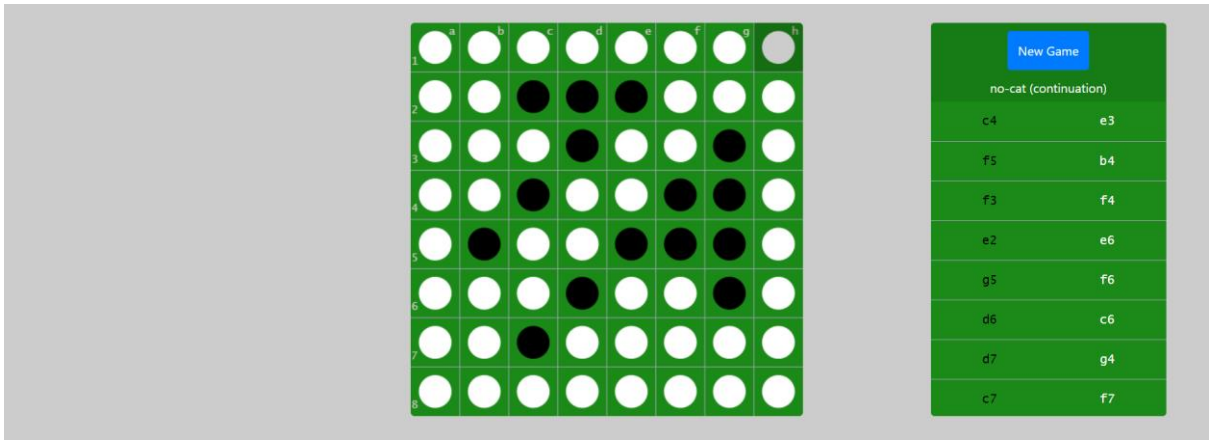
As demonstrated in the testing section, after each game has finished, the application displays an analysis table which appraises each move made in the game. This feature allows the user to identify which moves were mistakes and suggests better moves. Using this feature, users can identify by inference which aspects of their game (opening, midgame, endgame etc) are weakest. However, the application could be potentially improved in the future if the application actually calculated the relative strengths of the user's play during each game phase rather than relying on the user inferring this information from the analysis table by themselves. Nevertheless, both objectives H2 and H3 are satisfied by the solution.

H4. The program should be able to defeat a typical user on its maximum difficulty setting. In order to make this requirement measurable, I will propose that it should be able to defeat my mother, who is representative of the typical user.

In the final version of the program, the computer player is fairly strong and was able to defeat my mother, who represents a typical user, searching to a depth of 3. Two games were played and colours were swapped for the second game. The first game, in which the computer played black resulted in the following final position and was a win for the computer.



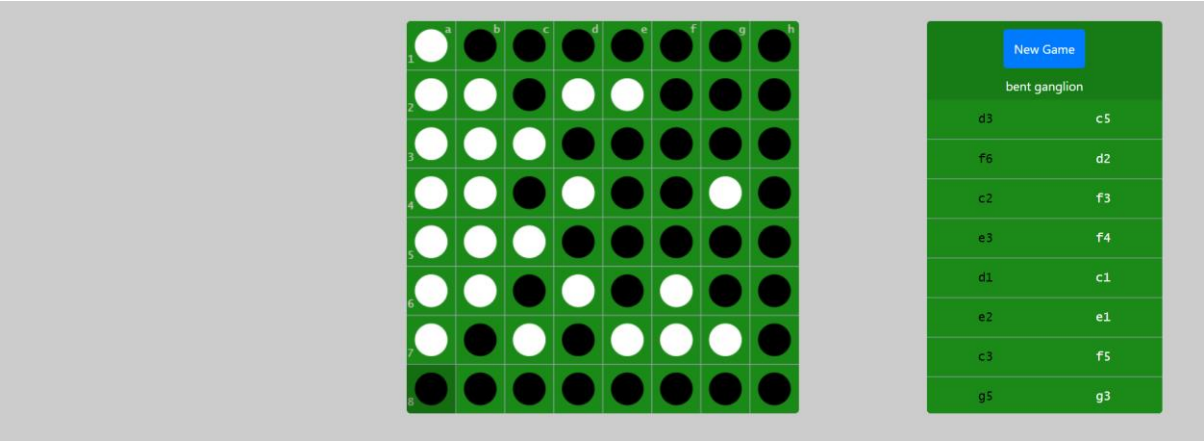
The second game, in which the computer played light, resulted in the following final position and was, once again, a win for the computer.



As demonstrated, even on a fairly shallow search depth of 3, the computer player was strong enough to consistently defeat the typical user. Therefore, it does achieve this objective. However, in order to further test the strength of the computer player, I used the application on a search of 4 to play against various online opponents via the website playok.com. A select few of these games are detailed below.

Game 1: Computer plays dark

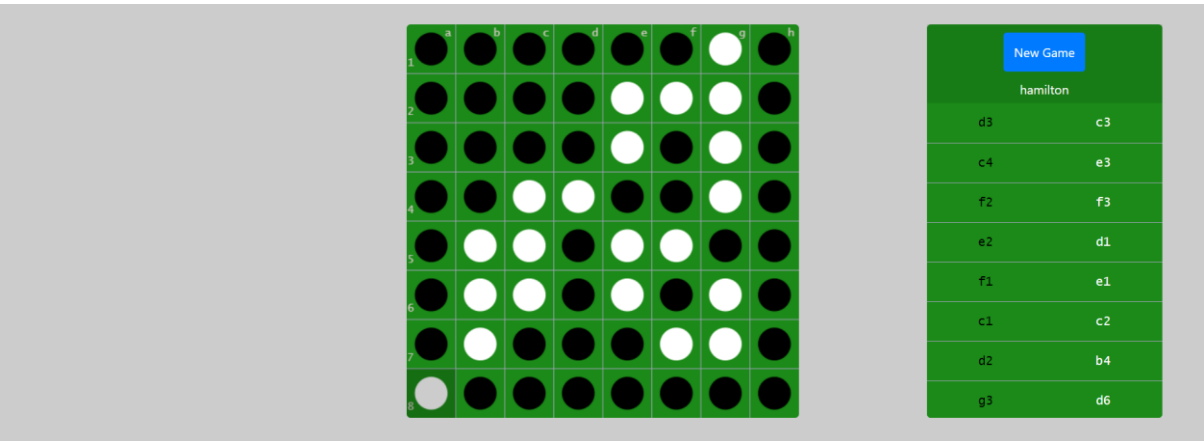
Transcript:
d3c5f6d2c2f3e3f4d1c1e2e1c3f5g5g3g6h5h6b4g4f2h4h3f1e6c4b3d6c7c6f7c8d7b5a4a6g2d8e7h1a5e8a7a3a2g1b2h2b6b7h7h8f8g8g7b1a1



Result: Dark wins 40-24

Game 2: Computer plays dark

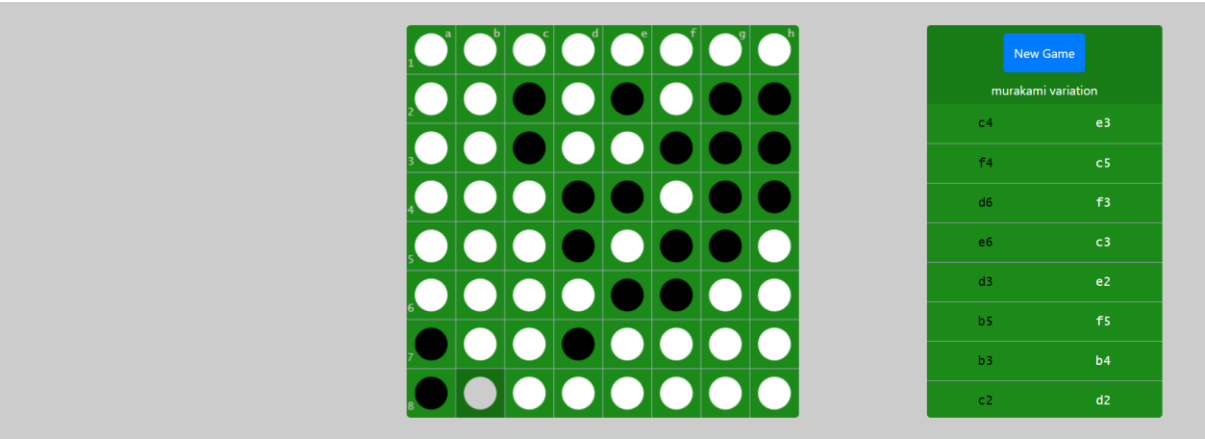
Transcript:
d3c3c4e3f2f3e2d1f1e1c1c2d2b4g3d6b3a2a4b2c5b1e7b5a1c7f4f5g6g4e6g5d7h3f6e8f7c6h6f8h4h5h2g2h7g7h1g1d8c8h8g8b8b7a5a6a3b6a7a8



Result: Dark wins 43-21

Game 3: Computer plays light

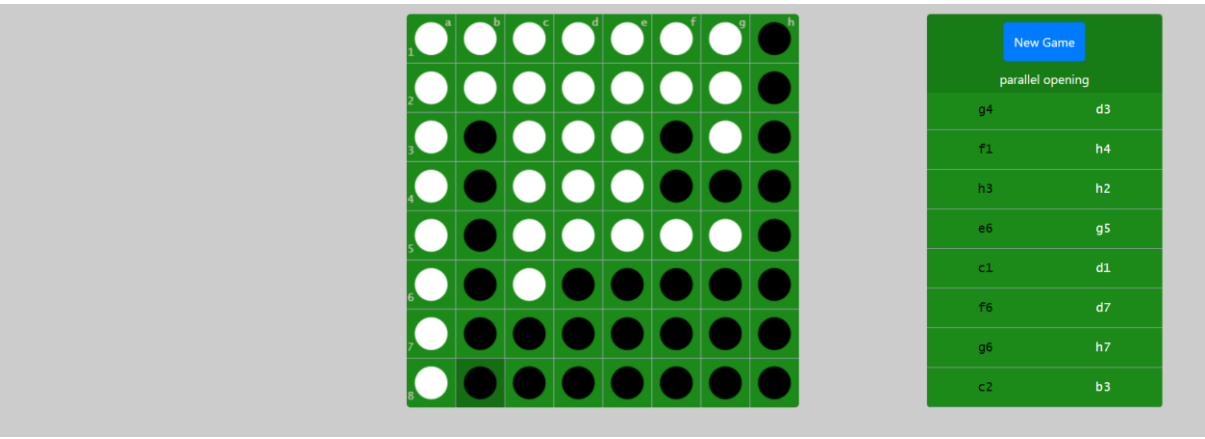
Transcript:
c4e3f4c5d6f3e6c3d3e2b5f5b3b4c2d2d1e1f2f1g6a3g5g4a5a4h3h5g3c1g2h1b2d7c6a1d8b1a2h6h2g7h7h8g8e7f6c7h4g1b6b7a8a6a7f8f7e8c8b8



Result: Light wins 44-20

Game 4: Computer plays dark

Transcript:
f5f4e3d2e2d6f3f2e1g3g4d3f1h4h3h2e6g5c1d1f6d7g6h7c2b3e7f8c8f7b2c7a4g2h5h6h1g1d8c3c4c5e8a1b4b1h8a2a3a5g8g7b5a6b7c6b6a7--a8b8



Result: Light wins 33-31

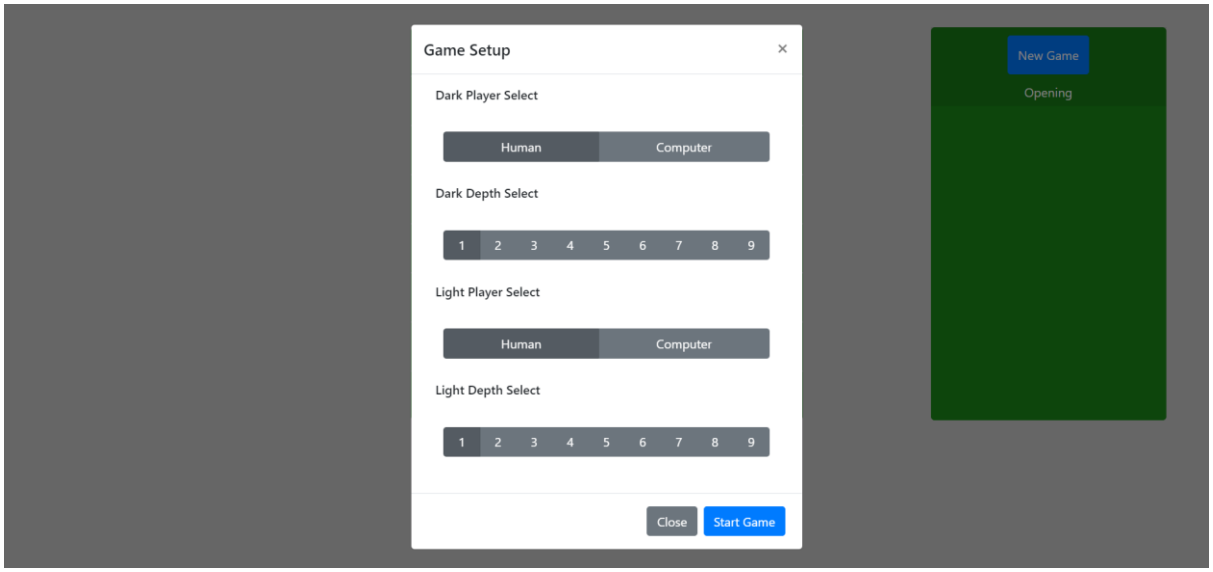
As demonstrated by these online games, the computer player performs well on a search depth of 4, only losing one game by a 2 disc margin. However, the application is not as strong as the strongest Reversi engines such as WZebra and Logistello. Such engines are capable of performing searches as deep as 12 moves almost instantaneously, whereas the largest depth to which the computer player in my application can search to in a reasonably short time is roughly 6. This difference is due to a combination of the vast precomputed tables used by these other engines, and the various heuristic techniques used to speed up their tree searches. As such, possible future enhancements to my application could include the implementation of search speed up techniques such as transposition tables, and the killer heuristic.

In addition, my Reversi engine could be made stronger by increasing the accuracy of its evaluation function. This could be achieved by adding additional factors to be considered about a board

position (e.g. stability), provided that this is worth the increase in computational expensiveness. Additionally, the weight of each factor could be refined, perhaps using a genetic method, rather than simply being determined by trial and error.

H5. The program should support the following match setups and allow the user to decide who plays light in each instance:
   a. Human vs Human
   b. Human vs Computer
   c. Computer vs Computer

As shown in the screenshot below, the final version of the solution does allow for all three of these combinations and so this objective has been achieved.



H6. The user should be able to interact with the program via a user-friendly graphical user interface.

In order to assess the degree to which this final objective has been satisfied, I consulted my mother, who represents the typical user, as to how friendly and intuitive the user interface is. In terms of the main display, the feedback that I received was that the board and right-hand panel are intuitively designed and that the colour scheme was effective. In terms of the game setup popup window, I received feedback that, although the use of radio buttons are effective in general, it is confusing that the option to select depth is presented even if the player selection is set to human. Therefore, a potential enhancement to the solution would be to set the depth selection buttons to only be visible when the player select option is set to computer.

Bibliography

Dewdney, A. *The New Turing Omnibus.* New York: Computer Science Press, 1993. Print.

Prasanna, N. *How to assign object properties to the result of an XMLHTTP Request. answer.* Available at https://stackoverflow.com/a/57307157/11868251. Accessed 01/08/2019

Lee K.-F., Mahajan S. *The Development of a world class Othello program.* Pittsburgh: Artificial Intelligence, 1990

Lee K.-F., Mahajan S. *BILL : a table-based, knowledge-intensive othello program*. Pittsburgh, 1986

Buro, M. *Statistical Feature Combination for the Evaluation of Game Positions.* Princeton: JAIR, 1995