

CS2006 H2: Othello

Team 21 Group Report

Overview

We have been tasked with recreating the board game Othello, using Haskell. This means our program must contain the same rules as the board game, such as what moves are legal, what tokens would flip after a move, and when a game is declared finished. To begin with, we have been given five starter Haskell files: Main.hs - where the game is initialised and the game loop can be found, Display.hs - where the board is rendered, Input.hs - where the user's inputs are handled, Board.hs - where the initial board and game state can be found and AI.hs - which contains the code for the AI that the user will play against.

Basic requirements have been met, as well as all easy and medium additional requirements and the second medium/hard and hard requirements.

Teamwork and Approach

The approach of the team towards the basic requirements was for all members to attempt to implement them as quickly as possible, with no element assigned to individual members, with the goal of completing these at an early stage to allow more time to be spent on the more difficult additional requirements. These additional requirements would then be assigned to different members of the team to implement individually.

Collaboration was achieved remotely by keeping code in a private GitHub repository and communicating via a group message chat to discuss the assignment of tasks and any issues being encountered, and to alert other members of the team when new features were committed to the repository.

Design and Implementation

Basic 1 - Game mechanics

The core mechanics of the game were quite complicated as each move had to enclose at least one piece of the opposite colour which then needed to be flipped, in addition to this we needed to check that the move was actually valid and in a space that wasn't already taken by another colour. Two of us came up with different implementations, one using a list of tuples and another using a hash map from the Data.Map module, we eventually decided to go for the maps approach as it had a more concise way of accessing pieces which would make other processes easier further down the line.

The Data.Map approach stored a map of Positions as keys and Colours as values, creating a board representation that allowed the lookup of the colour of piece in a given position in linear time rather than iterating through the original tuple list implementation, which was less efficient and less elegant. However, this approach would store values for keys that did not have a piece, i.e. empty positions, unlike the previous approach. Therefore, the Colour type had to be modified to also permit 'Empty' as a

colour. This approach of storing all positions, even empty ones, would allow the printing of the board to be much easier, since a value would be present that could be converted into a symbol for every position, not just those with pieces.

Core to the mechanics of the game is the flipping of pieces that are between two opponent pieces in a straight line. While easy for a human to identify these patterns, this proved more difficult to implement algorithmically. Once again, different approaches were attempted, with one team member attempting to write separate functions to check the piece to the left, right, top, bottom, etc. However, this was an inefficient approach that introduced a large amount of redundancy. Instead, a better approach seemed to be to implement a recursive function that could take a direction as a parameter, recursively check pieces in that direction until an empty piece was found, and generate a list of positions that must be flipped in that direction. A complete list of pieces to flip can then be curated by concatenating the result of mapping this function to a list of the 8 possible directions. This proved to be an elegant solution that could also be used to find the number of pieces flipped by a potential move by inspecting the length of the list of pieces to flip.

Performing the flips could then be done recursively as well, using the `Data.Map 'adjust'` method to change the colour value of the first position in the list, then calling itself on the tail of the list. This could, however, only be done after ensuring a move was valid, by checking that the destination was empty and would flip at least one piece.

Basic 2 - Print a textual representation of the game board

The textual representation of the board is generated in `Display.hs` through the use of `'intercalate'` with a series of list comprehensions. The colour of the tile is identified by looking up the position in the map, and the process is repeated for each column in a row, and then each row, before being intercalated together.

Basic 3 - Complete the implementation of the main game loop

To begin with we decided that the human player was always black, in this way we were able to come up with a simple game loop which first showed the `Gamestate`, then checked the colour, and if it was black it must be the human so it would ask for their move. This would cause the state to be updated to the other colour and the new board before looping. As we added detail to this process, one thing I noticed was that the `putStr "Move: "` line was sometimes out of order, I read up about this and discovered that output is buffered until you print a new line, and therefore this line needed a little help, so I added the `hFlush` handler function there. Once we had built the AI and had a working game, we implemented the `checkScore` and `gameOver` functions. In addition, we wrote a new function which would print the winner. We decided that it was useful to print the score after each go.

User input for each move is achieved using the standard `'getLine'` IO method, and this is processed in `Input.hs`. The x-axis letter input is converted into its numerical value by first converting it into its ASCII value using `'Enum'` methods.

Basic 4 - Implement a move generator and an evaluation function

This was the hardest part of the basic implementation, as it required us to create a function that not only searched for the best move currently, but one that was also able to traverse down possible future moves and get the best through that. I began by creating a simple AI that looked at the current list of moves and picked the head of the list. Once that worked, I looked at how I might do the tree traversal, I knew it had to be a recursive function, therefore I had to have a base case, when the depth was 0 and the rest of cases. I started by only having the base case. To do this, I implemented the evaluation function in Board.hs, this simply counted how many possible moves there were for that specific colour, then I used this to write a getScore function which would look at all the possible moves and worked out how many pieces that colour would have as a result of each move. I then used this to write a getMax function that would look at all the positions in the next_moves list and find the one that produced the highest score. I ran this and it seemed to work. Next, I moved on to the recursive part, this was challenging at first so I decided to begin by writing what would happen for depth = 1 and depth = 2 and then I could look at the similarities in order to create a more generic process. I decided the way this process would work is that I would traverse the tree by looking at the position that would produce the best score at each level, this required a slightly different getMax function since I had to store the position that had led to that group of moves as well as the moves. Once I was happy with this, I played the game a few times and it seemed to work sufficiently, however, later when we made it so that the board game size could be much smaller I noticed that I had made a slight error in that my algorithm caused problems if a move led to a pass at a lower depth, I fixed it by adding this check 'if (length moves) == 0 then getHardMove (depth - 1) tree', so that it essentially skipped it.

Later in development, avenues were investigated with the aim of improving the efficiency and quality of the AI's decisions. One way this was achieved was by substituting the original, simple 'gen' function, which offered the AI a list of every position on the board to build game trees from, with the already implemented hints method which, while still simple, could offer the AI up to five moves that flip the most pieces in the short-term. The AI can then build game trees and search them up to its given depth to evaluate which move to take. This improved the efficiency of the AI greatly, since it was now only investigating a maximum of five trees, rather than one tree for every position on the board, and the AI's behaviour changed too, behaving with a perceived improvement in skill.

Easy 1 - Add command line options to set up game options

For this requirement, I began by adding a single command line argument that allowed the user to choose either Black or White. Once I had worked out how to read in arguments, I needed a way to store which colour was human and which was the computer. The best way to do this seemed to be in the Gamestate record. Once I had done this, I decided to add an ability to change the board game size. This was more challenging as I had to change the way the board was printed. I also needed to think about how the default starting pieces would be placed on a different sized board. I changed the initBoard function so that it took a size variable that would then create a board of that specific size, then I created a function that would decide what the default starting values should be based on the size. I decided that the board size needed to be even, and that any size less than 4 would be too small and greater than 10 would be too big. This made the display easier to adjust as I could create a board for the largest size, and then use List.take to cut it to the right size. I decide to write a usage statement which would inform the user of what they needed to write.

As further features were added to the program, many were made additional command line options, and, to accommodate this, the processing of command line arguments was rewritten to avoid the many nested if-statements which comprised the original implementation. This is seen in the 'processArgs' method that uses anonymous functions to identify each argument, and is reused when changing the game options for a restart or mid-game.

Easy 2 - Implement Undo, to roll back the game state to after the previous move

In order to allow the player to revert the game state to after their previous move, the game state at the time of the previous move would have to be stored, with the most appropriate method for this being to store a previous state inside each 'GameState' structure. However, to avoid an infinitely recursive structure and to account for when no previous state exists, such as at the start of the game, it was decided that this would use the 'Maybe' monad. Therefore, each game state would usually store the previous state inside it, but, in the case where no such state exists, no previous state would be stored.

A function that would attempt to return this previous game state from a given state could then be written and used to restore the game to after the previous move. However, it was realised that rolling back once to the previous state would simply restore the game state to after the player had made their last move, leading to the AI making its move immediately. In order to achieve a real 'undo' the game state would have to be rolled back twice, to before the player had made their last move. Storing two nested previous game states seemed inelegant, therefore the reversion function had to be modified to include a counter parameter, allowing it to recurse once and revert the game state twice, to before the player's last move. If at any point the previous game state was 'Nothing', the function would simply return the existing game state. This allows the user to enter 'undo' instead of making their move to attempt to revert to before they made their previous move, and this feature is enabled in all games.

Easy 3 - Allow alternative starting positions

This requirement caused a lot of discussion within the group as to what exactly it meant. Eventually, having done some research on the original reversi game, we decided it meant that the first two moves for each player would not cause any flipping but did still have to follow the rule that they had to be touching an opposite colour in at least one direction. Once we had decided this, it was quite time consuming to implement as we had to create a whole new game loop for the start, since the makeMove method was different, however it wasn't actually too difficult since we already had a non-flipping approach stored in a previous version before we had implemented flipping. We decided to add it to the command line options.

Medium 1 - Allow options to be set in-game as well as at the command line

For this requirement, we decided only some options could be changed mid-game. These options being the colour that the user would play as and whether hints were showing. To change these options, we implemented a 'pause' command (which aided in a later requirement) where the user is asked if they wanted to change settings. If they chose to, they could re-enter the command line and if the only changes in said input were with the colour and hints setting, then said changes would come into effect.

Later, once a difficulty setting had been implemented, this was also added to the settings the user could change.

This requirement was originally implemented through the use of a very large collection of nested if statements, and had significant code repetition. It was therefore later entirely rewritten, incorporated into the processing of the player's move in the 'moveOptions' function. Instead of asking the user a series of questions to determine settings, a more streamlined approach seemed to be to allow the user to re-specify the command line settings in the same format as when they ran the program, allowing in-game options to be changed with one single step rather than miring the user in text based menus. These arguments were then passed to 'processArgs' as though they were the original command line arguments, but a boolean flag was used to indicate that the settings should be changed in place rather than starting a new game.

Medium 2 - Add the possibility of displaying hints

Before any version of hints or suggested moves could be implemented, it had to be decided exactly what form these hints would take. Determining the best move for the player would require a similar system of evaluation to that of the AI, searching game trees up to a set depth – a costly operation, and one that would have hints remove any need to the player to make a consideration, since the definitive 'best' move would be given to them. Instead, with efficiency and maintaining some element of human evaluation in mind, it was decided that hints would not be evaluated any deeper than the current turn, and that the program would provide the player with a selection of suggested moves, leaving them to choose which one to play, or to ignore these suggestions entirely. The suggestions would be generated by compiling a list of key-value pairs, where each key would be a possible move, and each value would be the number of opponent tiles that would be flipped by a move in that position, which could be achieved by zipping a list representation of all positions to another version of this list that had been mapped to the number of pieces to flip using the 'piecesToFlip' function from the logic in Board.hs. This list of pairs could then be sorted by these values, and up to five of the best moves would be suggested to the user in the prompt to make their move.

This function proved to be an elegant solution and was implemented in just two lines. The final list of up to five moves was then displayed to the user in the main game loop, using methods in Input.hs to convert the x-axis coordinate into a letter rather than a number.

Medium 3 - Implement a save game feature, and reload

It seemed that the best way to go about this was to store the game within in a text file. In the previous practical, one of the requirements was to read an input file containing a list of commands, therefore I had a basic understanding of reading and writing to files. I decided that if a user was reloading a saved game, they would want to have the exact same settings as before, therefore I decided to write in the command line arguments to the file, then the board. At first, I tried to write in the whole board to the file, however when I wanted to convert it back to a board from the text in the file this threw up an error. As a solution I decided to write in the list of pieces and the number of passes that had occurred separately, on the off chance that the user decided to save the game after one pass. Every time the user runs the game, the command line arguments are written to the file, however if they choose not to save

these are removed by writing an empty string to the file. I added in checks for if the user tried to reload a game when there wasn't one previously saved.

Medium to Hard - Implement time limits for moves and "Pause"

For the time limits, I started by playing around with the Data.Time library in Haskell, trying to record the time at the start and then wait until the current time was a certain number of seconds later. However, this was complicated and difficult, so I tried to find a better way. Eventually I came across 'hWaitForInput' which provided a very simple solution to the problem. I originally started with a 10 second timer, however, I found that this was a little too fast at times so I decided to make the time limit a choice for the user.

As stated before, the pause feature was actually implemented to aid with medium 1's game changing effects. It later turned out that this was a feature sought by this requirement. Thankfully, no code had to be changed from before and the second half of this requirement was instantly met.

Pausing and changing settings were later entirely rewritten to remove if statements and redundancy by making them part of 'moveOptions' and using 'processArgs', as previously mentioned.

Hard 2 - Implement multiple AIs with different skill levels and different strategies, and allow the player to choose

This requirement was the last to be considered, and multiple approaches for its implementation were considered by multiple team members. One investigated approach was that of using the depth parameter passed into the AI's move evaluation, and possibly allowing the user to modify this value to scale the difficulty of the AI. In theory, a higher depth value would have the AI evaluate each move's game tree further into the 'future' of the game, making its choice of move more informed and more likely to be the best possible move – creating a more difficult opponent for the player. However, when this avenue was investigated by playing through several games with different depth values set manually within the code, a correlation between a higher depth and a stronger AI was not as clear as expected. In fact, the AI would often be more successful on a lower depth setting, and could not compete with a human player using the hints to inform them regardless of depth. This testing suggested that the key factor in deciding whether the AI would win was whether the AI played first, rather than the depth value. Higher depth values also caused AI moves to take much longer to evaluate, and, in the case of very high values, caused elements of the program to be executed out of order. Therefore, this approach was not taken further.

The design choice we eventually went with was difficulty being decided by the AI making the most or least flips possible as this has a closer correlation to perceivable difficulty than any other option. The problem with this is that in certain situations of Othello, it is more advantageous to have less tokens on the board than the other player as that means you have a wider range of places to place your own tokens and cause flips. Therefore, making more flips at the start of a game on a bigger board may be less effective than making moves that cause less flips. This means that in some situations the user can win out against the hard AI relatively quickly if making their own most effective moves, whilst losing to the easy AI. However, overall, the hard AI won many more games against the testers and came close to winning even more times, so it rightfully deserves the title of the 'hard option'.

Testing

Command Line Testing

Input	Expected Output	Actual Output	What This Shows
./Main White 10 hard	Usage message	<pre>mod6@pc3-019-1:~/Documents/cs2006/H2/h2-master/src \$./Main Usage: -> To reload saved game: program reload -> To start new game: program [<colour> <board size(4/6/8/10) mal-start> <hints-on/hints-off> <easy/hard>] Run without arguments for default settings</pre>	Othello does not start if the correct number of arguments aren't in the command line.
./Main White 10 default- start hints-on hard	Game starts, asking the user for the time limit	<pre>mod6@pc3-019-1:~/Documents/cs2006/H2/h2-master/src \$./Main White 10 default-start hints-on hard Controls: When prompted to move, enter the co-ordinate e.g. D4 You can also type: undo -- this will get rid of the most recent move restart -- this will restart the game, and give you the choice to change settings pause -- this will pause the game, and give you the choice to change settings quit -- this will end the game, but give you the choice to save it first What time limit would you like (seconds) (10-60)?</pre>	Othello successfully starts when the correct number of arguments are in the command line for a new game
./Main reload	Game is loaded again, asking the user for the time limit	<pre>mod6@pc3-019-1:~/Documents/cs2006/H2/h2-master/src \$./Main r eload Controls: When prompted to move, enter the co-ordinate e.g. D4 You can also type: undo -- this will get rid of the most recent move restart -- this will restart the game, and give you the choice to change settings pause -- this will pause the game, and give you the choice to change settings quit -- this will end the game, but give you the choice to save it first What time limit would you like (seconds) (10-60)?</pre>	A saved Othello game is successfully loaded when the specific reload command line is used

Time Limit Testing

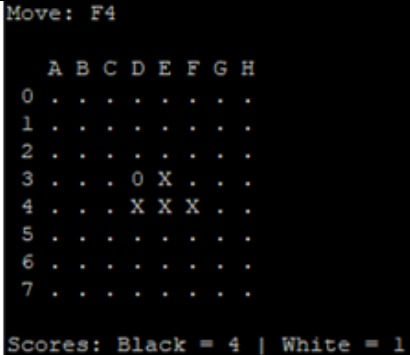
Input	Expected Output	Actual Output	What This Shows
5	Error message	<pre>What time limit would you like (seconds) (10-60)? 5 Invalid time! Controls:</pre>	Exceptional times are not allowed
30	Board is loaded and the game can begin	<pre>What time limit would you like (seconds) (10-60)? 30 X = Black O = White A B C D E F G H I J 0 1 2 3 4 0 X 5 X O 6 7 8 9 Scores: Black = 2 White = 2 Black to move Suggested moves: ["G5","F6","E3","D4"]</pre>	Normal times are allowed
60	Board is loaded and the game can begin	<pre>What time limit would you like (seconds) (10-60)? 60 X = Black O = White A B C D E F G H I J 0 1 2 3 4 0 X 5 X O 6 7 8 9 Scores: Black = 2 White = 2 Black to move Suggested moves: ["G5","F6","E3","D4"]</pre>	Extreme times are allowed

First Turn Testing

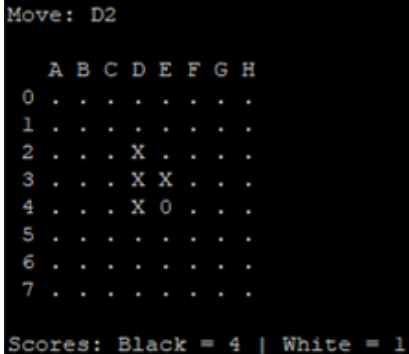
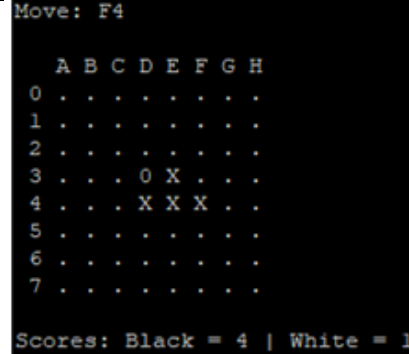
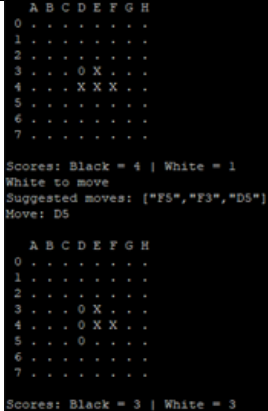
Input	Expected Output	Actual Output	What This Shows
'Black' in command line	Player is to make the first move	<pre> A B C D E F G H I J 0 1 2 3 4 0 X 5 X 0 6 7 8 9 Scores: Black = 2 White = 2 Black to move </pre>	If the user is the black colour, they will make the first move
'White' in command line	AI makes the first move	<pre> A B C D E F G H I J 0 1 2 3 4 0 X 5 X 0 6 7 8 9 Scores: Black = 2 White = 2 Black to move A B C D E F G H I J 0 1 2 3 4 0 X 5 X X X . . . 6 7 8 9 Scores: Black = 4 White = 1 </pre>	If the user is the white colour, the AI will make the first move.

Legal Move Check from Start

Input	Expected Output	Actual Output	What This Shows
A0	Move is not allowed	<pre> Move: A0 Error, please replay </pre>	Any position is not a valid move
D5	Move is not allowed	<pre> Move: D5 Error, please replay </pre>	Positions beside the turn taker's own tokens are not valid moves.
D2	Move is allowed and black token appears in that position	<pre> Move: D2 A B C D E F G H 0 1 2 X . . . 3 X X . . 4 X 0 . . 5 6 7 Scores: Black = 4 White = 1 </pre>	Moves beside the other player's tokens that are in a line with the turn taker's own tokens are allowed

F4	Move is allowed and black token appears in that position	 <pre> Move: F4 A B C D E F G H 0 1 2 3 . . . 0 X . . . 4 . . . X X X . . 5 6 7 Scores: Black = 4 White = 1 </pre>	Moves beside the other player's tokens that are in a line with the turn taker's own tokens are allowed
----	--	---	--

Token Flipping Test

Input	Expected Output	Actual Output	What This Shows
D2	White token found between the tokens will flip.	 <pre> Move: D2 A B C D E F G H 0 1 2 . . . X 3 . . . X X . . . 4 . . . X 0 . . . 5 6 7 Scores: Black = 4 White = 1 </pre>	Tokens flip when they connected two tokens of the other colour vertically.
F4	White token found between the tokens will flip.	 <pre> Move: F4 A B C D E F G H 0 1 2 3 . . . 0 X . . . 4 . . . X X X . . 5 6 7 Scores: Black = 4 White = 1 </pre>	Tokens flip when they connected two tokens of the other colour horizontally.
D5	Black token found between the tokens will flip.	 <pre> A B C D E F G H 0 1 2 3 . . . 0 X . . . 4 . . . 0 X X . . 5 . . . 0 6 7 Scores: Black = 3 White = 3 White to move Suggested moves: ["F5", "F3", "D5"] Move: D5 </pre>	Tokens flip when they connected two tokens of the other colour vertically.

F3	Black token found between the tokens will flip.	<pre> A B C D E F G H 0 1 2 3 . . 0 X . . 4 . . X X X . . 5 6 7 Scores: Black = 4 White = 1 White to move Suggested moves: ["F5","F3","D5"] Move: F3 A B C D E F G H 0 1 2 3 . . 0 0 0 . . 4 . . X X X . . 5 6 7 Scores: Black = 3 White = 3 </pre>	Tokens flip when they connected two tokens of the other colour horizontally.
----	---	--	--

AI Making Moves Test

Input	Expected Output	Actual Output	What This Shows
A0	Move is not allowed, and user is asked to enter a valid move	<pre> Move: A0 Error, please replay </pre>	AI isn't called if the user fails to make a valid move.
D2	Move is allowed and the AI responds by making its own valid move	<pre> Move: D2 A B C D E F G H 0 1 2 . . X . . . 3 . . X X . . 4 . . X 0 . . 5 6 7 Scores: Black = 4 White = 1 White to move A B C D E F G H 0 1 2 . . X 0 . . 3 . . X 0 . . 4 . . X 0 . . 5 6 7 Scores: Black = 3 White = 3 </pre>	AI is called after the user makes a valid move, making a valid move of its own

Passing Test

Input	Expected Output	Actual Output	What This Shows
Move that calls the AI to make a move, leaving the player no places to place a token	The player passes	<pre> Move: B0 A B C D 0 0 X X X 1 0 X X . 2 0 X X X 3 . . X 0 Scores: Black = 9 White = 4 White to move Black passed. A B C D 0 0 X X X 1 0 X X . 2 0 X X X 3 . 0 0 0 </pre>	When there are no moves to be made by the player, the player passes

Move that leaves the AI with no places to place a token	The AI passes	<pre> A B C D E F G H 0 1 2 . . X . . X . 0 3 . . . X X X X 0 4 . . . X X X X 0 5 X . 0 6 X X X 7 Scores: Black = 14 White = 4 Black to move Suggested moves: ["H1"] Move: H1 White passed. </pre>	When there are no moves to be made by the AI, the AI passes
---	---------------	---	---

Game Ending Test

Input	Expected Output	Actual Output	What This Shows
Move that results in both player's passing	The game ends	<pre> A B C D E F G H 0 1 2 . . X . . X . 0 3 . . . X X X X 0 4 . . . X X X X 0 5 X . 0 6 X X X 7 Scores: Black = 14 White = 4 Black to move Suggested moves: ["H1"] Move: H1 White passed. Black passed. Game Over Black wins! </pre>	When there are no moves to be made either player, or both players pass consecutively, the game ends.
Move that places a token in the final free position on the board	The game ends	<pre> Move: I9 A B C D E F G H I J 0 0 0 0 0 0 0 0 0 X 0 1 0 0 0 0 0 0 0 0 X X 0 2 0 0 0 0 0 X 0 X 0 X 0 3 0 X 0 X X X X 0 X X 4 0 X 0 X 0 X 0 X 0 5 0 X 0 X 0 0 X X 0 6 0 X 0 X 0 X 0 X 0 7 0 X 0 0 0 0 X 0 X X 8 0 X X X X 0 X 0 X 0 9 0 X X X X X X . X 0 Scores: Black = 43 White = 56 White to move Game Over White wins! </pre>	When there are no moves to be made because the board is full, the game ends.

Additional Requirement Testing

Input	Expected Output	Actual Output	What This Shows
./Main White 10 default-start hints-on hard	The game is loaded on a board of length 10. The AI has gone first as the user is White.	<pre> A B C D E F G H I J 0 1 2 3 4 . . . X 5 6 7 8 9 Scores: Black = 2 White = 0 Black to move A B C D E F G H I J 0 1 2 3 4 . . . X 5 6 7 8 9 Scores: Black = 4 White = 1 White to move </pre>	The user can make decisions about the board size and colour that they play as from the command line. Easy 1 has been implemented.

./Main Black 6 default-start hints-on hard	The game is loaded on a board of length 6. The player is to go first as they are Black.	<pre> A B C D E F 0 1 2 . . 0 X . . 3 . . X 0 . . 4 5 Scores: Black = 2 White = 2 Black to move </pre>	The user can make decisions about the board size and colour that they play as from the command line. Easy 1 has been implemented.
undo	The game is set back to the user's previous move, which is the game start.	<pre> A B C D E F 0 1 2 . . 0 X . . 3 . . X 0 X . 4 0 . 5 Scores: Black = 3 White = 3 Black to move Suggested moves: ["E5","D4","C1","B2"] Move: undo Attempting to revert to previous game state... A B C D E F 0 1 2 . . 0 X . . 3 . . X 0 . . 4 5 Scores: Black = 2 White = 2 Black to move </pre>	The user can revert to their last move using the undo command. This can be done continuously to, essentially, start the game again. Easy 2 has been implemented.
undo	The game is set back to the user's previous move.	<pre> A B C D E F 0 1 2 . . 0 X . . 3 . . X 0 0 0 4 X . 5 X . Scores: Black = 4 White = 4 Black to move Suggested moves: ["B1","F4","E2","D4","C1"] Move: undo Attempting to revert to previous game state... A B C D E F 0 1 2 . . 0 X . . 3 . . X 0 X . 4 0 . 5 Scores: Black = 3 White = 3 Black to move </pre>	The user can revert to their last move using the undo command. Easy 2 has been implemented.
./Main Black 6 custom-start hints-on hard	An empty board should appear to the user allowing them to place the first of two starting tokens down	<pre> A B C D E F 0 1 2 3 4 5 Black to move </pre>	The user can opt to start with a blank board. Easy 3 has been implemented.
./Main Black 6 custom-start hints-on hard A0	After the empty board appears, and entering in the first coordinate, one token appears on the board	<pre> A B C D E F 0 1 2 3 4 5 Black to move Move: A0 A B C D E F 0 X 1 2 3 4 5 </pre>	The user can opt to start with a blank board, placing two of the four starting tokens themselves. Easy 3 has been implemented.
./Main Black 6 custom-start hints-on hard A0	After the empty board appears, and placing a token on the board, the AI also places one token on the board.	<pre> A B C D E F 0 1 2 3 4 5 Black to move Move: A0 A B C D E F 0 X 1 2 3 4 5 White to move Move: B1 A B C D E F 0 X 1 2 3 4 5 </pre>	The user can opt to start with a blank board, placing two of the four starting tokens themselves whilst the AI places the other two. Easy 3 has been implemented.

pause	The menu to change settings appears	<pre> A B C D E F 0 1 2 . . O X . . 3 . . X O . . 4 5 Scores: Black = 2 White = 2 Black to move Suggested moves: ["E3", "D4", "C1", "B2"] Move: pause Game stopped. Would You like to change game options? (y/n) </pre>	The game recognises the pause command. Medium to Hard 2 has been implemented.
pause y Black 6 default-start hints-on hard	Hints are now on	<pre> A B C D E F 0 1 2 . . X . . 3 . . X . . 4 5 Scores: Black = 2 White = 2 Black to move Game stopped. Game stopped. Would You like to change game options? (y/n) y Note - only player colour and hint settings can be changed without restarting the game Your previous settings were: "Black & default-start hints-off hard" Please reconfirm command line arguments Black & default-start hints-on hard A B C D E F 0 1 2 . . X . . 3 . . X . . 4 5 Scores: Black = 2 White = 2 Black to move Suggested moves: ["E3", "D4", "C1", "B2"] </pre>	The game can change hint settings mid game. Medium 1 has been achieved.
pause y White 6 default-start hints-off hard	The user is now White, meaning the AI is Black and has thus made the first move.	<pre> A B C D E F 0 1 2 . . X . . 3 . . X . . 4 5 Scores: Black = 2 White = 2 Black to move Suggested moves: ["E3", "D4", "C1", "B2"] Game stopped. Would You like to change game options? (y/n) y Note - only player colour and hint settings can be changed without restarting the game Your previous settings were: "Black & default-start hints-on hard" Please reconfirm command line arguments White & default-start hints-off hard A B C D E F 0 1 2 . . X . . 3 . . X . . 4 5 Scores: Black = 2 White = 2 Black to move Suggested moves: ["E3", "D4", "C1", "B2"] </pre>	The user can change colour and hint settings mid game. Medium 1 has been achieved.
./Main Black 6 custom-start hints-on hard	All possible first moves are shown to the user	<pre> A B C D E F 0 1 2 . . O X . . 3 . . X O . . 4 5 Scores: Black = 2 White = 2 Black to move Suggested moves: ["E3", "D4", "C1", "B2"] </pre>	Hints are shown to the user about possible moves at the start of the game. Medium 2 has been achieved.
Multiple Moves into the game	The five moves that would produce the most move flips should be shown to the user	<pre> A B C D E F G H 0 1 2 . . X . . X . . 3 . . . X O O . . 4 . . . X X O X . 5 X . 0 6 X X X 7 Scores: Black = 11 White = 5 Black to move Suggested moves: ["H3", "H2", "G2", "E2", "D2"] </pre>	Hints are shown to the user about possible moves during the game. Medium 2 has been achieved.
quit	The option to save should appear when the user tries to quit	<pre> Move: quit save? (y/n)? </pre>	The game understands the quit command
quit Y ./Main reload	The game state from before the game was saved and quitted should reappear	<pre> A B C D E F G H 0 1 2 . . X . . X . . 3 . . . X O O . . 4 . . . X X O X . 5 X . 0 6 X X X 7 Scores: Black = 4 White = 4 Black to move Suggested moves: ["H3", "H2", "G2", "E2", "D2"] Game stopped. Game stopped. Would You like to change game options? (y/n) y Note - only player colour and hint settings can be changed without restarting the game Your previous settings were: "Black & default-start hints-on hard" Please reconfirm command line arguments Black & default-start hints-on hard A B C D E F G H 0 1 2 . . X . . X . . 3 . . . X O O . . 4 . . . X X O X . 5 X . 0 6 X X X 7 Scores: Black = 4 White = 4 Black to move Suggested moves: ["H3", "H2", "G2", "E2", "D2"] </pre>	The user can quit a game and opt to save the game state. Then the user can reload the saved game state later. Medium 3 has been implemented.

No input for 10 seconds (the specified time limit)	The player's turn should be passed	<pre> 10 X = Black O = White A B C D E F G H 0 1 2 3 . . . O X . . 4 . . . X O . . 5 6 7 Scores: Black = 2 White = 2 Black to move Suggested moves: ["F4","E5","D2","C3"] Move: Out of time! Black passed. </pre>	Each turn now has a time limit that if the user does not meet, their turn is passed. Medium to Hard 2 has been implemented.
Easy AI chosen All best possible moves (thanks to hints)	The user should beat the AI by a decent sized margin	<pre> A B C D E F 0 O X X X X X 1 O O X O X X 2 O X O X O X 3 O X X O X X 4 O X X X X X 5 O . X X X X Scores: Black = 24 White = 11 White to move Suggested moves: ["B5"] Move: B5 Game Over Black wins! </pre>	At the easy difficulty, the user should find it that if they make the right moves, they can beat the AI comfortably. Hard 2 has been implemented.
Easy AI chosen All best possible moves (thanks to hints)	The user should lose to the AI by a close margin	<pre> A B C D E F 0 X X X X X X 1 O X X X X O 2 . X X X X O 3 X X X O O O 4 X O O O O O 5 X O O O O O Scores: Black = 19 White = 16 White to move Game Over White wins! </pre>	At the hard difficulty, the user should find that by even making good moves, they can find it difficult to beat the AI. Hard 2 has been implemented.

Compiling and Running this Solution

Compilation of this program can be done through two methods – using cabal or ghc.

Before compilation with either method, please ensure cabal's package list is up to date by running 'cabal update', and please ensure that the System.Random module is installed by running 'cabal install random'.

To compile this program with ghc, navigate to the 'src' directory and run 'ghc Main.hs'. This will create the executable file 'Main'.

To compile this program with cabal, remain outside of the 'src' directory and execute 'cabal build'. This will create an executable called 'Othello' in the 'bin' subdirectory within the 'dist' directory that cabal will create.

The program can be run from the relevant executable. This executable can be run without command line arguments to use the default game settings, or custom settings can be supplied as follows:

```
./<executable> [ <colour> <board-size> <custom-start/normal-start> <hints-on/hints-off> <easy/hard>]
```

If a game has been saved before quitting, the last save game can be reloaded by running the program as follows:

```
./<executable> reload
```

Evaluation and Summary

Overall, this program has certainly helped made us much more aware of the advantages and disadvantages of Haskell, and we have all come away with a much greater understanding of the language. Creating a board game using Haskell has been an enjoyable experience, and we have all developed our skills in Othello, as a result.

We would have liked to improve the graphics using ncurses, as this may have allowed us to get rid of the co-ordinates all together and let us click on the square, and also make the board more colourful which would be nice. However, there was not enough time and we felt that making sure the game works well was more important. Additional routes for possible extension of this project include improvements to the AI, possibly implementing a more nuanced means of scaling the AI difficulty.