Group report:

This report details the completeness of our Othello program with regards to the basic and additional requirements, the design and implementation of the program and any known problems with the final program. In addition to this report, each member has submitted an individual report detailing their team contributions, summary of provenance stating which fragments of code they wrote and any individual problems they experienced and how they solved them. Furthermore, some basic testing for our program is contained in the submitted file "`Testing.pdf`".

Summary of functionality:

Our program has completed all the basic, easy and medium specifications. We have also added network play and multiple AI difficulties as a hard extension.

Basic:

- Implement game mechanics in `Board.hs`
    - Our basic game works with placing and capturing pieces. Our main method to capture pieces is to check all 8 directions and generate a list of positions in those directions for pieces that should be flipped. The lists are generated by looping through until we either get to the end of the board in that direction or we see a piece that is the same colour. If it gets to the end without finding a piece of the same colour then an empty list is returned as not pieces are flipped, otherwise it stops once it sees a piece of the same colour and returns a list of all the positions before it stopped. Then with all 8 lists from the 8 directions, we flip every piece in those positions.
    - For placing pieces, we simple check that that position does not already contain a piece and that the move will cause pieces to be flipped by checking the length of the `posList`.
- Implement the graphics in `Draw.hs`
    - We had drawing with gloss pictures by drawing a large square as the board and lines to split each positions. The pieces were just filled circles that were either black or white. But since we have moved drawing to use bitmaps, none of the code for drawing gloss pictures is in the final submission.
    - See bitmaps
- Implement appropriate event handlers in `Input.hs`
    - When clicking, we first check that the place clicked is not outside the board. Initially we had a bug with players being able to place pieces outside of the drawn board so this check prevents that.
    - Next we have to use either the `makeMove` function or `startMove` function.

`startmove` is used for reversi start, which allows players to choose where to put their 2 initial pieces at the start of the game.

- The functions `snapX` and `snapY` convert the mouse coordinates that are clicked on screen to grid coordinates that are used in our back end. If the move clicked is valid, `handleInput` will return the world with the new move made.
- Similarly we have guards for the different keys a player could press to change the options in game or do in game actions such as restart the game or save the game.

- Implement a move generator and evaluation in `AI.hs`
  - Our move generator simply returns the list of available positions. It is difficult to generate only moves that are good in the generator, so how good moves are is determined in the evaluate function and bad moves are not filtered out in the move generator as the aim of this method is to find all the possible moves, not the best.
  - Our AI Yusuki uses a minimax tree search to get the best possible move. He assumes the opponent will also make the best possible move for them according to his algorithm.
  - Yusuki's evaluation function is a heuristic. We give the corner positions a higher value and the positions next to corners a negative value. The sides of the board also get a slightly higher value. Otherwise each piece he controls gets one value. Losing the game also gives an extremely negative value, so he will always try to avoid losing if he can see the conclusion of the game. This is so that Yusuki will prioritise taking the corners and sides, while forcing his opponents to take the positions next to the corners. These values are rather arbitrary but Yusuki seems to do really well during our play testing, whereas he wasn't doing well before we added these heuristics to his evaluation. If we had more time we would have liked to of added different versions of Yusuki with different weightings for different styles of play.

Easy:

- Add command line options
  - We pass in command line options using `getArgs` in the main function of our game. These `args` are then passed to `initWorld` where the function `setArgs` will set up the initial world based on the arguments that were passed.
  - A detailed list of arguments and their functions can be found in the README of the submission.

- Implement Undo
    - To implement undo turn we store a list of previous states in the world. These states store the important information we need when we want to revert back to a previous turn such as the board, whose turn it is and the timers. Whenever we make a new move, that move is appended to the important information from the world of that move is stored in the `stateList`.
    - If the list of states is empty then we cannot go back any further. Currently we allow the player to go back all the way to the start of the game.
- Alternate start (reversi)
    - For reversi, we allow the players to choose where to place their first 2 pieces in the centre of the board. This is slightly separate from the rest of the game, so the game actually starts when both players have placed their 2 initial pieces down.
    - Here we need a different function checkStart to return the list of possible moves while the players are making the reversi moves because they are different and do not require pieces to be flipped when being placed. The pieces are only placed in the four positions in the middle of the board.

Medium:

- Set options in game
    - To set options in game we extend the input to include keys from the keyboard. Then we can return a world with the options changed based on the key clicked and also some other parameters. For example we do not allow undoing when playing over the network because that would be cheating against your opponent.
- Bitmap images
    - For bitmaps, we create a new data structure which will store all the bitmaps at the start of the program. Then this structure can be passed around the various drawing functions to provide the bitmap those functions need.
    - We can then draw the game based on the board we received from the world by passing the list of positions and colour to our draw functions to draw accordingly. `Draw.hs` also has lots of helper functions that help translate where to draw the pictures from the positions of the board to the actual drawing and help scale the bitmaps for different sized boards.
    - Because we were unable to draw bitmaps in 32 bit due to a compression error,

we could not have transparent bitmaps and draw pieces on top of the board. So our bitmaps for the pieces have the board as their background so when they are all drawn it makes up the entire board. They are still drawn over the empty board pieces which makes the drawing slightly inefficient, having to draw twice as many pieces by the time the board is filled.

    - This also means the order in which we draw matters, because the pictures that are drawn first will be overlapped by the pictures that are drawn afterwards, this is why the pieces are drawn after the board.
    - If we had more time we could have added alternate bitmaps for pieces so players could choose what their pieces look like.

- Save game and load game
    - To save and load the game, we use `Data.Binary` and `Data.ByteString` to serialise the data structures into a bytestring. The world is saved out into a file "`save.othello`" which can also be loaded once saved.
    - This serialisation is also used to send moves over the network, but we do not send the whole world because it contains some information that we do not want to pass to the other player such as which colour is the networked player.

Hard:

- Timer and pause
    - A timer for each player is displayed on either side of the board. The timer is stored in the `World` record in milliseconds for each player and decremented by ten every time `updateWorld` is called as `updateWorld` is called every ten milliseconds. If either player's time reaches zero then the game enters a game over state and the player who ran out of time loses.
    - For pausing the we add a new boolean field to the `World` record which determines whether the game is paused or not. Input from the 'p' key toggles the pause state. If the world is in a pause state then `updateWorld` does not change the world at all and a pause screen is drawn. While in a pause state, the only action a player can carry out is resuming to prevent players doing things like undoing the game state while the board is not visible due to the pause.
    - Timer is disabled for play over network and for AI players. Over the network when waiting for the other player to make their turn, a game instance blocks until it receives a turn. While it is blocking nothing else can be processed so `updateWorld` cannot decrement the timer. Similarly, when an AI is making their

turn the program is processing possible moves to make so nothing else can be processed.

- Network play
    - For Network play we extend the `World` record further to include a `Socket` and a boolean to determine whether a player is using the client version of the game or the server. This `Socket` is a `Maybe Socket` and is `Nothing` if the game is not being played over the network. This allows our functions an easy way to tell if the game is being played over the network or not as some functions such as `updateWorld` must send data over the network but only when the server is on.
    - To set up network play one player must first start the game with the “-server” argument and may set any number of additional arguments normally. Any AI arguments that are set are ignored as both players must be human over the network. The client must then start the game with the “-client” option and specify the host name after that to connect to. The server then sends the starting world with all its settings to the client so that they're both starting with the same world. After that, at every move the game instance making a turn sends the board created from its move to the other player. The game instance not making a turn waits until it receives a new board from the other player and updates their world with the new board to allow them to then make their turn.
    - When using network play the server is always black and the client is always white. Obviously this is not ideal and with more time we would add arguments to set whether the server and client's colours.
    - Saving, loading, undoing, pausing and timers are disabled over the network as they each cause their own issues when combined with the network.
- Different AIs with different strategies
    - In addition to Yusuki, we implemented a random AI. It simply picks a random move out of all possible moves so is a less challenging opponent. This allows players to choose an easier AI to play against if they wish. The random AI can also be pitted against Yusuki to demonstrate his prowess in playing Othello.

Known problems:

- When using network play the server is always black and the client is always white. Obviously this is not ideal and with more time we would add arguments to set the server and client's colours.
- Unfortunately, we did not have time to add full error handling for the network in unexpected cases such as disconnects. If these cases do occur a default error is printed from the socket. With more time we would produce meaningful error messages on these occurrences and stop the game from crashing when the errors

occur.

- While it is not a players turn and they are waiting (either for AI to make a move or other player over network) anything they input will be buffered and all processed as soon as it is their turn. This is problematic as it means a player can actually make their move in advance by accidentally clicking while waiting. Even worse, a player buffering multiple moves over the network while waiting can cause more than one valid move to be processed and sent causing both players to become completely out of order. We were unsure how we could fix such issues and so they remain in the final version.
- If two AI players play against each other, one or the other will always be processing their possible moves to make their turn. While processing possible moves nothing else can be processed so the game is unable to draw every move unless it can find time between AI players passing turns which is very quick so is not always the case. This means it is very difficult to view a full game of AI versus AI and often the game cannot draw until the game over screen.