Haskell 2 -Report

Overview:
The purpose of this practical was to implement the game of Gomoku in Haskell. All of the basic requirements have been completed. Furthermore, these additional requirements have been completed. Command line flags have been added to allow customisation of the game. Undo was also implemented. Different rule variants were also added to the game, there was an option for the standard Gomoku, a handicapped game, where the rules are as standard, but an implementation of the extra rules of three and three and four and four, and the 'Pente' variation of the game was also implemented. With regards to the AI, both of the AI's algorithms were based upon the MiniMax algorithm, with 2 different evaluation algorithms, one where the focus of the algorithm was based on attacking moves, and one where the focus was based on defending defending moves. The same evaluate functions were al and so used to give hints to the user. Save and load functionality was also added to the game. Time limits were also implemented for human players in the game. In Game options were also added to the project, with the ability to be able undo a move, pause the game, save the game, and load a game. Network multi player was also implemented, so the user was able to play gomoku across the network. Furthermore, Bitmapped images were also added instead of using the gloss library to draw the images on the board. Running Instructions are stored in the ReadME. Haddock was also used to build documentation for the project.

List of Requirements Completed:
All Basic Requirements,
Command Line Options
Undo
Rule Variants
In Game Options
Bitmapped Images
Hints
Save/Load Game
Time Limits and Pause Game
Pente
Multiple AI's with different skill levels and strategies
Network Play

Design and Implementation

Game Mechanics and Options

There were 2 main areas of the game mechanics which were implemented, the ability to check whether a move being ma and de is legal by the rules which the game is being played, and the ability to check whether the player has won or not.

There were 3 different options for checking whether the move being played was legal or not depending on what rule was being played. Due to the way the draw function was implemented, the make move function never got called unless the position was valid. This meant that the function didn't have to check whether the position to be played in was within the board or not. When selecting the rules, the only difference between the standard and the handicap rules was the extra rules of three and three and four and four. However, for Pente, there was an extra rule which was stored which held the number of captures for black and white, and it also stored a list of the board after every move, which was done owing to the need to be able to undo moves in the case of a capture, something which would not be able to be done using the standard undo version.

For the standard rules, all the function had to check was whether there was already a piece there, which was done by simply filtering the list of pieces on the board with the position which was going to be inserted, and if there was already a piece there, the make move function returned nothing to show that play was invalid.

For the second option, when the extra rules of three and three and four and four, the same rules were also applied, along with 2 other rules. The interpretation of the rule which we implemented was to not allow the black piece to be played such that in the playing of the piece, 2 rows of four and/or 2 unbounded rows of three could not occur simultaneously by placing of that one piece. To implement this, the getDirectionList function was called, which gets a list of numbers representing the string of pieces of the same colour in the four directions, North - South, East – West, SouthEast -NorthWest, and SouthWest – NorthEast, this meant that to check that there were less than 2 directions in which there were 4 pieces in a row in which the last piece played was a part of. To implement the rule of three was slightly more complex, in that the values had to be unbounded at both sides. The definition of unbounded which we took to mean was there had to be an empty square either side of a row, i.e. a row which starts at the end of the board is not unbounded as it can only have at most one empty square at the end of the row. To find all rows of three, the function called the get direction list function and then filtered out the rows of three. To then check whether the rows were unbounded or not, for each row, the function found the end of the row by going in the direction of the row, i.e. kept going North/East/NorthWest/NorthEast depending on what direction the row of 3 occurred in until it found the end of the row and then it checked whether a) the squares beyond both ends of the row were within the board, and then if not running the function to see if there was a piece was already there on the board, and only if there were 2 rows of 3, which were both unbounded, was the move disallowed.

Pente is a variation on Gomoku that introduces the possibility for pieces to be captured. A capture occurs when a stone is placed on the board in such a way that two opposing stones adjacent to each other (horizontally, vertically or diagonally) are ‘flanked’. These stones are then removed from the board, and the game proceeds as normal. Pente allows players to win in one of two ways – the traditional method used in Gomoku whereby a series of stones of the same colour are placed adjacent to each other to create a run of a target size, as well as a new method where a player can win if they achieve the same target number of captures.

The rules of Pente are not dissimilar to Gomoku, and they did not require the main game loop to be modified significantly to implement them. A new function named playRule was introduced to accommodate different Rule types (where Rule was defined as Rule = Standard | Handicap | Pente PenteState). Based on the rule being played by the current game, this function ran different tests to calculate the board that would be produced by placing a piece in a specified position. For Pente, this logic was split into two functions. The first, capturePente, checked a specified direction leading out from the most recently played piece to check whether it had resulted in two opposition stones being flanked. It if had, then these stones were removed, and the resulting board was returned. The second function, updatePenteState, made use of foldr to run capturePente across every possible direction and return the board produced by any resulting captures. If the standard target had been met by placing adjacent pieces or making sufficient captures, then the game was declared to have been won.

Undo:
Due to the fact that pieces were stored as a list within board, whenever a new piece was added, it was added to the front of a list, thus the list became a stack containing the order the pieces were added in. Thus undo was a simple operation for the Standard and Handicap rules, as when playing local multiplayer, all that was required was to change the pieces on the board to the list minus the

first element in the stack. However, when playing against the AI, the list had to be reduced by 2 to take back the AI's move as well as the users.

Pente, however, was significantly different – owning to the need to record captures and ensure that they were replaced in the correct way when a move was undone. When playing against the AI, there was the possibility that removing two pieces from the head of the list of pieces would unintentionally remove a piece that had been initially played before the two pieces most recently captured. As a result, it became necessary to implement a more complicated undo mechanism – whereby the entire board was recorded at the start of each turn. This allowed the board state to be easily reverted by simply popping the most recent board from the list of boards in the PenteState object included in the Rule data type with the Pente constructor.

In Game Options

In game options were also added to the functions by typing characters into the functions, the functions to undo a move as above was 'u', 'n' started a new game by generating a new world with the same options as the old one. 'p' paused the game by stopping the countdown timer if it is your move, to make the game fair, such that the user wouldn't use pause to analyse their next move and to stop the timer, all of the pieces were hidden on the board until the game was un-paused.

Checkwon:

To check if the game has been won, the get directionlist function was called which got a list of pieces in the 4 directions, North - South, East – West, SouthEast -NorthWest, and SouthWest – NorthEast. It did this by calling the checkfunction, which took the piece and the list of 8 possible directions from the piece just played, and then recursively called itself, passing the position it had + 1 in the direction, and checked if a) there was a piece there, and b) if the piece there was the same colour as the starting piece to match, it did this until the new piece didn't match, at which point it returned with the number of recursive calls made which represented the number of pieces in that direction. This function was called with the starting piece and all of the possible 8 directions, which generated the list of pieces going North, South, East,West etc.., to then get a list of the number of in the 4 directions, since the way the list was set up, the 8 directions were in order, e.g. the list went [NorthNumPieces, SouthNumPieces, EastNumPieces, WestNumPieces, etc] thus to get a total for the 4 directions, all that needed to happen was to sum up the numbers in a list 2 at a time, i.e. end up with a list which was [NorthNumPieces + SouthNumPieces, EastNumPieces + WestNumPieces], then 1 needed to be subtracted from each element as in all directions, the initial piece was counted, so when summing the directions, the initial piece was counted twice, thus 1 was subtracted to get the actual piece count.

When this function was implemented, check win was easy to implement, after playing a piece, the check win function was called to see if a win occurred by playing the last piece, by calling the getdirectionlist function and then seeing if any of the four directions equalled the target exactly, since having a row longer than the target did not constitute a win.

Save Load

The game has an inbuilt gameplay option for saving an loading. Pressing the 's' key will write the binary encoded current world into a file named "save". Pressing the 'l' key will read the encoded world from the 'save' file, decode it and replace the current games world with the decoded world object. We had to replace the gloss play function in the main loop with playIO to allow for this functionality as the binary serialization functions used returned IO World objects.

Command Line Options

The command line options to the game were passed in using command flags, and then they were parsed into the function as required, while they changed the values of the options of the game. To

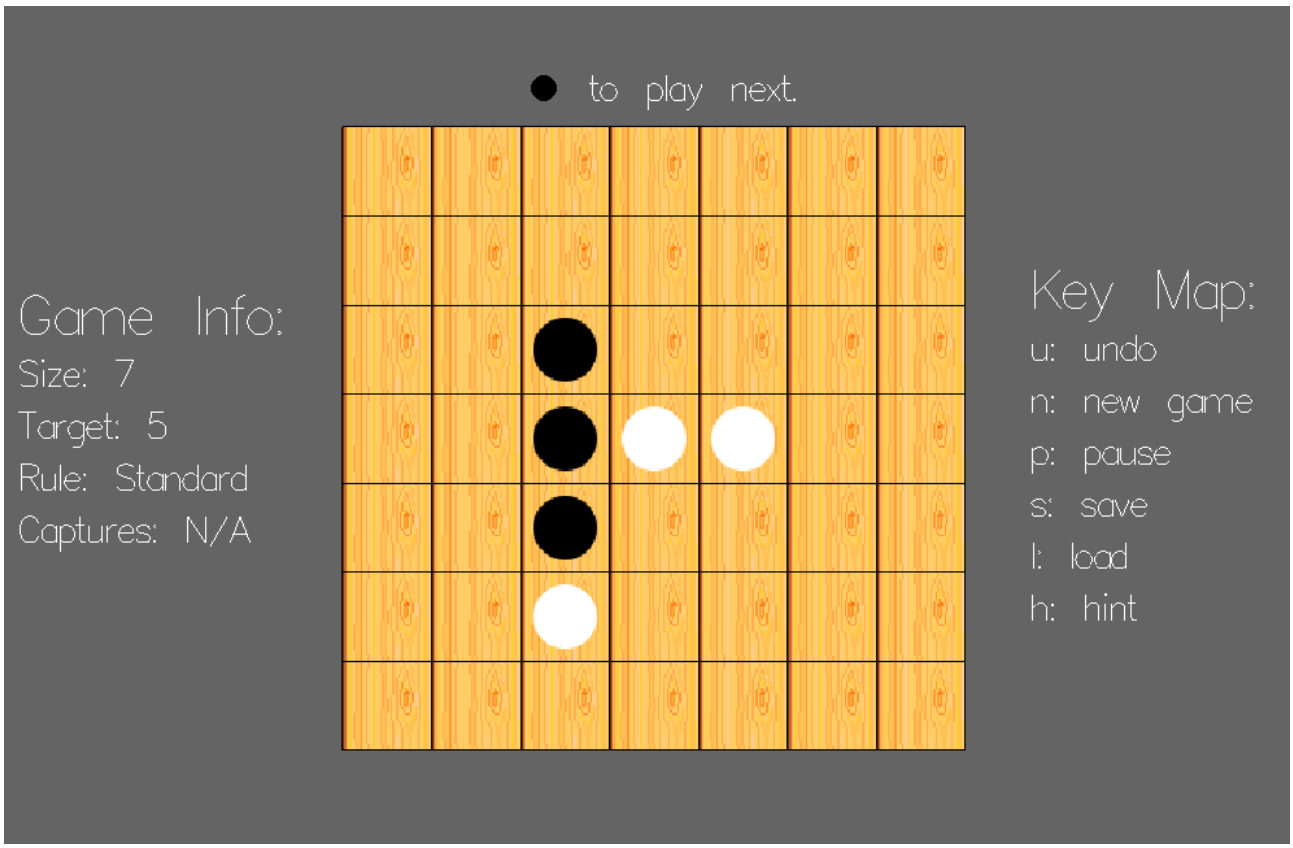
do this, the inbuilt function ‘Option’ was used where it took each command line flag, and then matched it to an option and returned the value that the user entered into the function. To set all of the options, all of the command line arguments were taken in, and then a foldl was used to modify the game options based on what options were specified by the user.

Drawing the Board

The graphics used to display the board were split into five different layers:

1. boardLayer – this layer contained all pictures used to display the base of the board, which were the bitmap images used to represent individual board squares (tiled to construct the board itself).
2. pausedLayer – this layer contained all the pictures that depended on whether the game was paused at any one time. These included the bitmap images used to represent the black and white pieces placed on the board itself, and the text that appeared to inform the user when the game was paused.
3. infoLayer – this layer contained all the pictures that informed the user of the state of the game, as well as key bindings for the various actions that could be triggered during the game.
4. gridLayer – this layer contained the grid of lines that delineated the boundaries between the squares placed on the board. These were placed over the top of the rest of the images on the board to ensure that they clearly formed the grid boundaries, and were not rendered underneath any other pictures.
5. WonLayer – this layer contained the pictures that were used to display either that a particular player had won, or otherwise which player was set to play the next turn. These appeared in the form of text at the top of the board.

A screenshot taken from the game is given below:
and



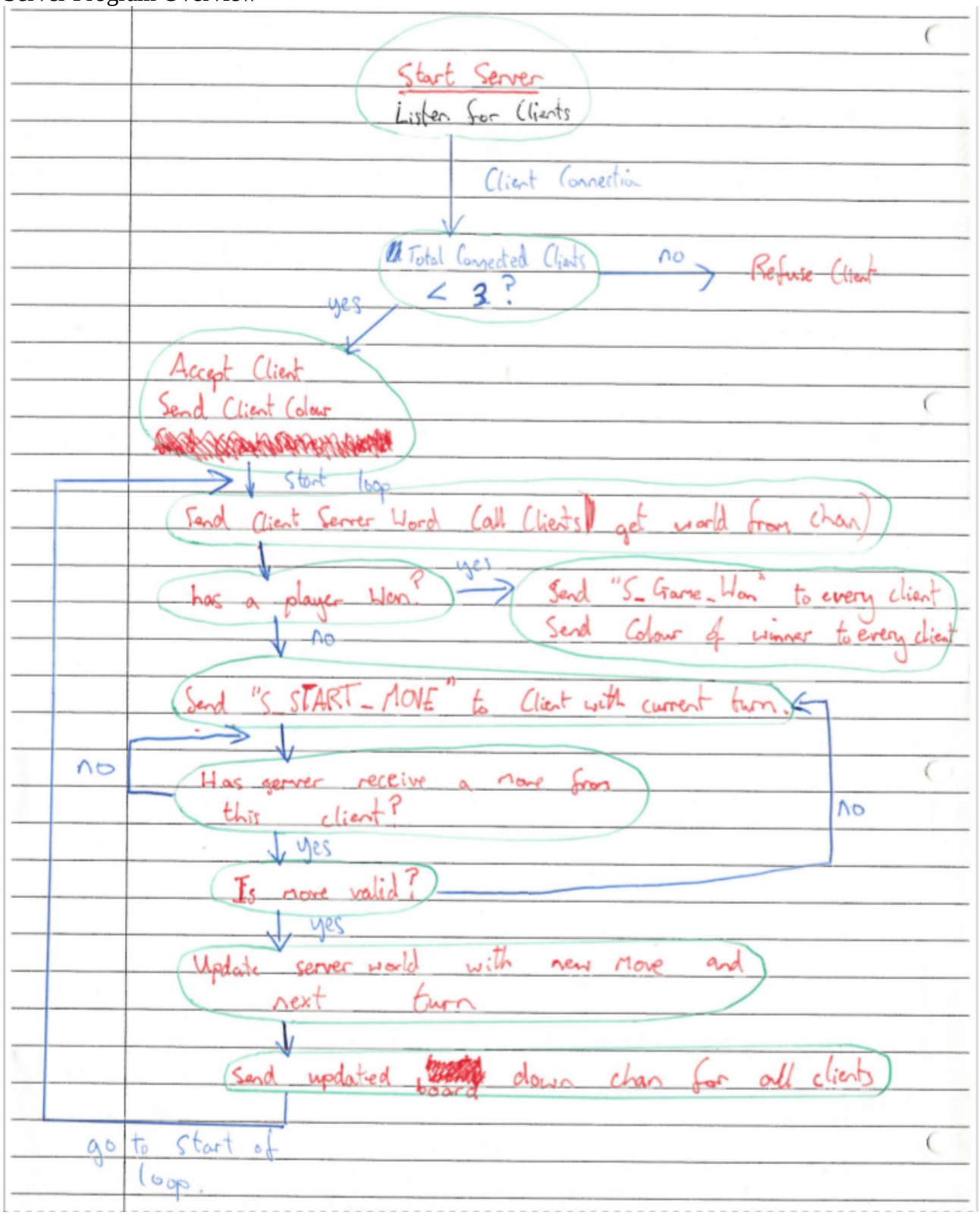
AI

For the main AI we implemented minimax to allow for a smart AI that would make calculated moves. We went straight fro the minimax implementation to cover both a basic AI and the hard advanced AI. There are multiple versions of the AI as the evaluate function and depth can be changed to allow the AI to favour certain strategies more and to allow the AI to look forward more moves. The default is in my opinion the best AI, the aggressive AI. The AI is aggressive but also makes calculated defensive moves. The evaluation function calculates a score as follows: For every unbounded line made by the AI add 4^(size of the line) * (size of the line) to the score. For every unbounded line made by the other player subtract (4^(size of the line) * (size of the line)) * 3 from the score. This allows the AI to make aggressive moves while also actively trying to block the other player. The second advanced AI instead has a similar evaluation function but multiples the other player subtraction by 10 instead to make it more actively defensive. We decide to add a simpler AI later to have a much more basic AI to compare the advanced AIs against. This basic AI doesn't take advantage of the generated gameTree in the same way as the other two do. The AI detects unbounded lines made by the other player (length target – 1 and target – 2 mostly) and actively attempts to block the other player. This works most of the time but it is fairly easy to beat. The minimax Ais work much better but it is nice to see a contrast. We should also mention that the minimax Ais detect won boards at any depth of the minimax recursion tree (before trying to evaluate the board at the leaf node). In these cases we returned a score of maxBound for the AI winning and minBound for the other player winning. I minus the current depth from this value to allow for wins at higher points of the tree to be valued higher. This means that the AI doesn't miss a winning move looking too fair down the tree. We unfortunately didn't get to implement alpha beta pruning as we did not have time but instead we did code a function that makes the AI only play moves in small radius of the currently played pieces. This saves time at the start of the game but doesn't really save much time when pieces are spread far over the board. The AI has been tested by multiple people and seems to work very well.

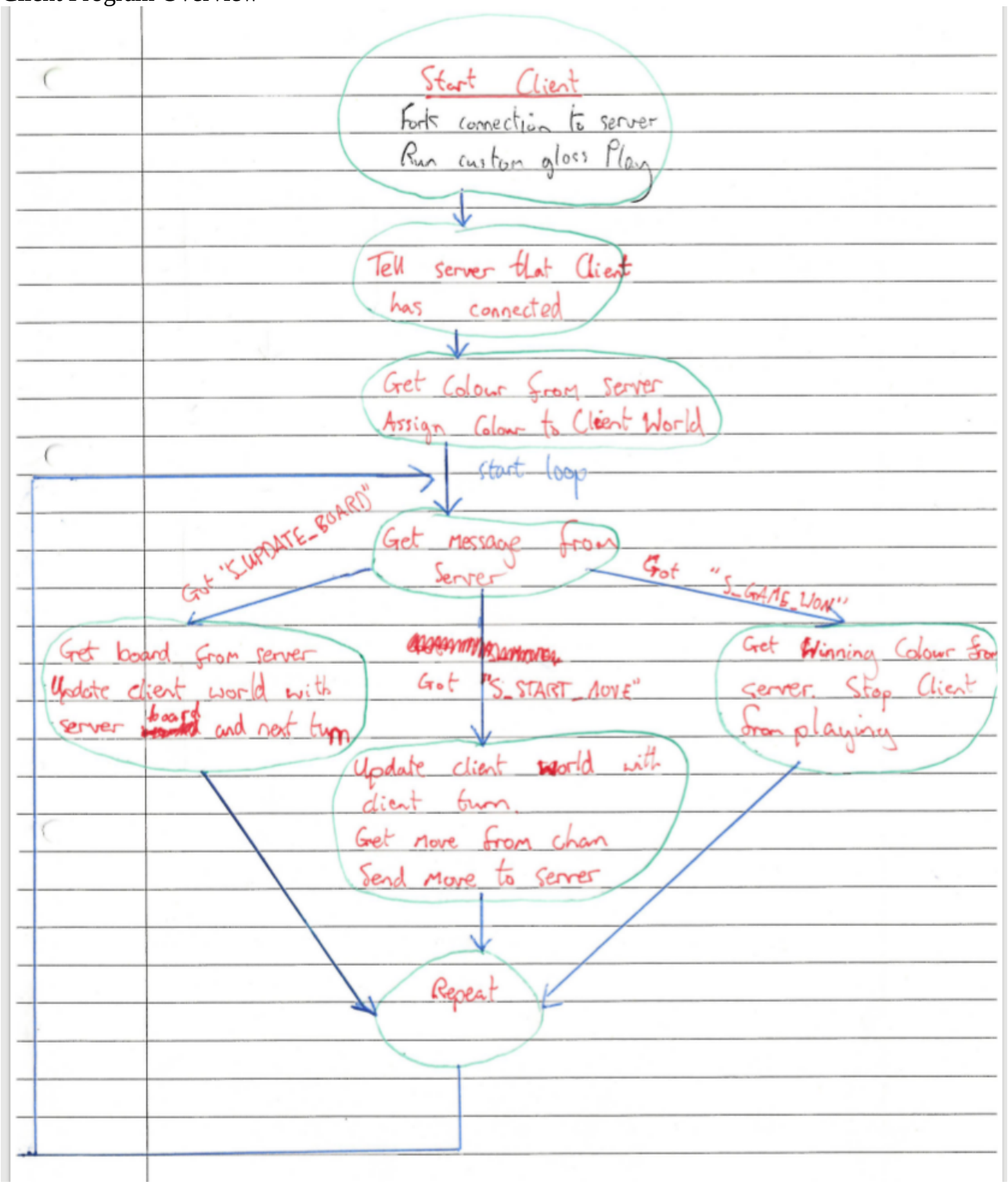
Network Play

The network play works by having two clients with their own world variables that is displayed on each execution by gloss. The Server holds the actual world variable used to play the game. The server assigns each client a colour and sends the server board on connection. When the clients get sent a colour they update their “playerColour” field with the sent colour. When the clients get sent a board they update the board of their local world variable with the sent board to be displayed using gloss. The server asks the first connected client to make a move. When a client is asked to make a move the client reads a move from the chan. There is a click event listener function “handleClientInput” that sends the clicked move through a chan that the connection loop reads from. Once the move is read from the chan it is sent to the server after telling the server that a move is about to be made. After this the client listens for the next message (start move, update baord or game won). Once the server receives a move it will make the move on the server board, change the turn and send the updated world through the server dupChan. The dupChan is brought into the server loop as an argument so the next iterator of the loop can read the updated world. This allows all connections to update the same world. We used a dupChan to allow for to potential extra players in the future. On the next iteration of the server loop the server reads from the dupChan to get the upadated world with the new board and next colour. The server can now continue the same flow as already described by sending the updated board to the next client and then asking the client to make a move. Once a move is sent to the server the makeMove function is run to update the board. If the move is not valid then the server sends a reject move message. This happens when for example client 1 places two pieces before client 2 places one and the second piece client 1 place happens to be the same piece client 2 placed. At this point the server will reject client 1’s second piece placement. This is easy enough to handle but we didn’t not get time to handle it full unfortunately. When the server receives a move and makes it, it checks to see if the updated board is won. If it is the server sends a win message to the clients and then sends the winning colour. The clients catch this message, stop play and display the winner. The multiplayer is quite separate from the normal game and so it is only recommended to use the port and ip address command line arguments when running network play as to not mess with the default settings of the world. Most of the in game key functionality has been removed to due to multiplayer development happening at the same time as development of functions like save/load/undo/new game. The hint button can be used in network play. Below are diagrams of the general flow of the server program, the client program and a flow of how they communicate with each other to help with explaining our implementation. The server and clients had to use TVar variables to allow for atomically updates. This means there is no change of corruption by two clients updating at the same time. We had to edit the playIO gloss function to take in the TVar world variable at each client to be brought be used to display the world. This TVar world is then used in the connection loop.

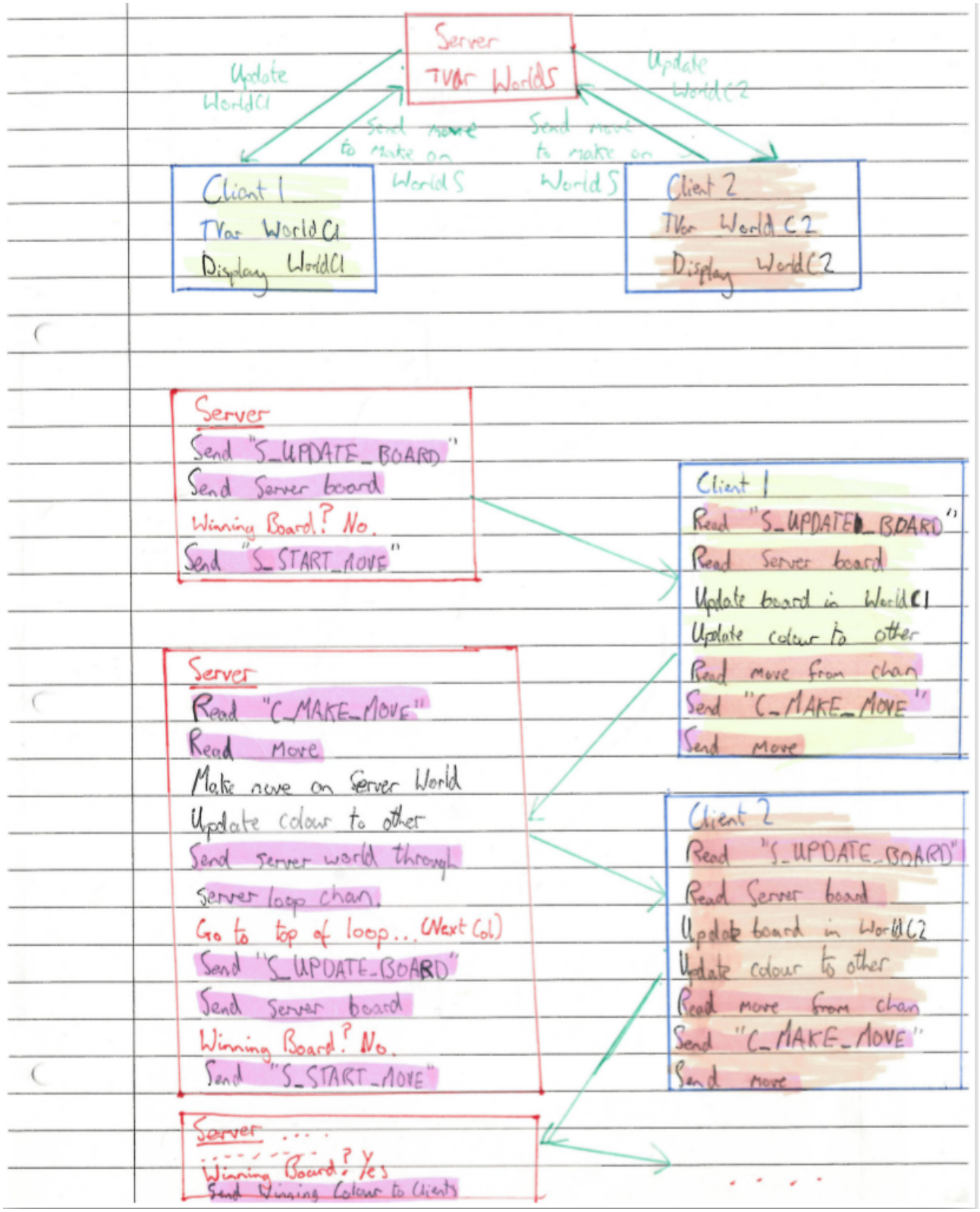
Server Program Overview



Client Program Overview



Communication Overview



and Problems

The majority of problems experienced during the development of the board mechanics were type checking problems, where the program didn't return the correct types and thus the program didn't compile, however when those had been resolved, there weren't any significant logic errors.

There were however logical problems which occurred while developing the code for the AI and Networking. For the networking, one of the hardest problems occurred in miscommunication between Client and Server, as the server would be expecting something the client wasn't sending across at that time, and the client waiting for the server to respond, resulting in the program hanging. Another problem was that you needed to fork each client connection to allow them to connect to the server.

With regards to the AI, implementing the Minimax algorithm 'properly' was difficult initially due to the recursion making the dataflow of the program more difficult to understand. One of the other major problems was in order to make the AI play better, it was realised that wins occurring earlier on in the gametree should be valued at more than wins lower down in the gametree.

There was also problems getting the bitmapped images to load as expected, eventually basic bitmapped images had to be loaded and used.

One of the other major problems was experienced with Laziness of Haskell, where some functions which we though would get evaluated didn't so we couldn't see what the problem was.

Testing

There was extensive testing performed on the project to check that it worked as expected in all areas of the game, and features implemented worked as expected.

Contribution

Conclusion

In conclusion, we completed the basic requirements of the specification, and extensions were also completed to allow network play, allow playing against different AI's, and further customisation of the game by allowing command line options to the game. Furthermore, there were ingame options available to the user, undo the last move played, to get hints, pause and play the game, start a new game, and save and load. Also, multiple AI's were written, and Network Play was also implemented. We worked well as a team, accomplishing all of the tasks, and were able to co-ordinate our efforts well.