2018/19

Student Name: **Callum Clarke**

Student ID: 201170688

Project Title: Developing the game of Checkers.

Supervisor: **Sven Scheve**

DEPARTMENT OF COMPUTER SCIENCE

The University of Liverpool, Liverpool L69 3BX

Contents

Abstract.

The main goal of this project was to create the classical game of Checkers (Also known as English Draughts) that can be played on a traditional desktop/laptop or on a mobile device (such a smartphone or tablet).

The game has two versions which can be played, these being a solitaire version and a multiplayer version. The solitaire version involves the user playing a game against the computer. Within this game the computer decides which moves to make based on a 'difficulty' level decided before the game starts. There are three levels of difficulty, these being 'easy', 'medium' and 'hard'. Depending on which difficulty is selected a different set of logic that decides the moves the computer makes will be used.

The multiplayer version of the game allows for two separate users (On different machines/devices) to play a game. This works by one user first creating a game room by clicking the 'host game' button on the website. They are then shown a code which is tied to and unique to the game room that was just created. They can then share this code with the person they want to play with via any communication channel they have between each other. That person can then click the "join game" button on the website at which point they will be prompted to enter the code for the room they want to join. Once the code is entered both users will be within the same game room and the game will be allowed to start. The user that first created the game room (The 'host' of the room) will make the first move.

Both versions of the game include all the same features needed in order to properly facilitate a game of checkers. For example, both versions of the game include a recursion algorithm that correctly identifies all the moves that a selected piece can make. This includes moves that involve the piece making a jump over an enemy piece and moves that require multiple jumps (hence the use of recursion for this algorithm). Another example would be the creation of king pieces which happens when one of the user's pieces makes it all the way to the starting row of their opponent.

This project has been quite interesting to carry out and develop especially when developing the multiplayer version of the game. NodeJS and Socket.IO made developing the networking side of the game very easy as these libraries are very well suited for small networking uses such as this project. Overall, I would say the project has been a success as both versions of the game have been finished and work correctly and allow for a game to be played.

1. Introduction.

This project set the goal of creating the game of checkers that would be able to be played on a computer (such as a desktop or a smartphone, etc). The problem set out by this project is the creation of the algorithms for facilitating the actual functionality of the game and the implementation of the networking side of the multiplayer aspect of the game (such as the handing the creation and sending of packets of data).

The main aim of the project is to have two working and playable versions of the game (a solitaire version and a multiplayer version). The solitaire version of the game should allow the user to have a game of checkers against the computer with the computer player having a set difficulty level. These levels should change the behaviour of the computer player in some way that either makes it easier or harder to play against. The multiplayer version of the game should allow two separate users to play a game of checkers against each other from two different machines.

One of the main objectives of the project is to develop the algorithms needed to be as efficient as possible in terms of computing. This will be achieved by not repeating code that is essentially the same thing and trying to make the algorithms created as efficient as possible.

1.1. Project Challenges.

This project had many challenges that needed to be overcome. The main challenge I feel this project presented was regarding the development of the different difficulties that needed to be programmed into the computer player for the solitaire version of the game. This proved particularly hard to program and implement into the game as the solution I originally settled upon implementing (The use of the MiniMax algorithm (University, n.d.)) didn't end up working correctly with my code and eventually led to me not using MiniMax at all. Instead I settled with implementing my own (admittedly much simpler and not as complex) logic for the three difficulties.

Another challenge that this project presented was developing the networking side of the multiplayer version of the game. Since the multiplayer version of the game required two separate users to be on distributed machines when playing against each other, this required someway of transferring data (such as the state of the grid for example) between the two computers. This challenge didn't end up being as big of a problem as anticipated. This is mainly due to the use of the libraries NodeJS and Socket.IO. Node.JS

was very useful as it allowed me to set up a server for the game which essentially acted as a middle man for data passing between the two players.
The use of a server also allowed for multiple games to be played at the same time and NodeJS facilitated the handling of data within these games (though the use of a feature called namespaces which will be talked about in more detail later). Socket.IO was the library that allowed for packets of data to be sent between any users that were connected to the server.

One other challenge that presented itself when developing the multiplayer aspect of the game was deciding how long to keep a game room open (still existing in memory on the server) when one of the two players had lost connection.

1.2. Project Solution / Final Software.

The solution for this project comes in two parts. The first part is a standalone website that fulfils the requirement of the user being able to play a game of checkers against the computer. The second part comes in the form of a server that can be executed on a machine. This server will then accept connections from users and allow those users to link with each other and play a game of checkers. A user connects to the to the server though a web browser just as they would connect to any normal website. This part fulfils the requirement of two users being able to play a game of checkers together from distributed machines.

Both implementations are developed in the same programming language (JavaScript) and both use the same rendering solution for the grid of the game (Normal HTML Tables and CSS).

1.3. Effectiveness of the Final Produced Software.

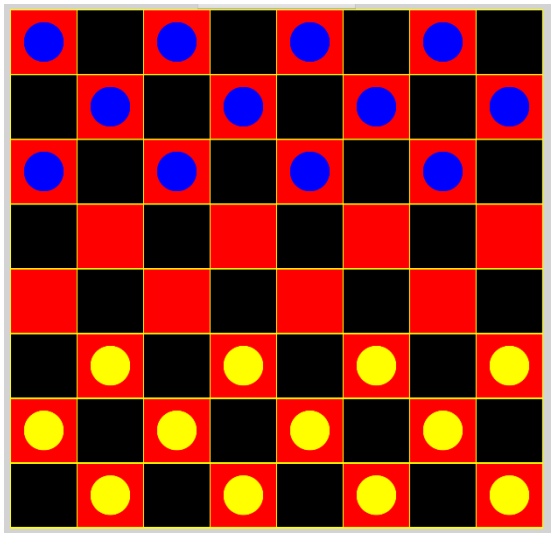
The main code for both solutions has been rewritten multiple times. This is mainly due to unforeseen problems that came up during development that required a whole rework of the system in order to get it working correctly. The final solutions developed I feel achieve the goal set by the project of creating two playable versions of the game of checkers.

The solutions produced may not be the most elegant in terms of code design or efficiency but they both achieve the goal of the project of facilitating a game(s) of checkers.

2. Project Background.

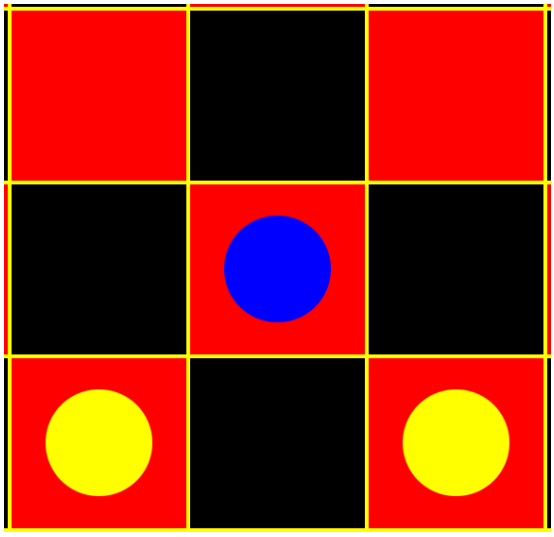
English draughts is an adaption of the game of standard draughts (Checkers) where the size of the grid and the number of pieces on the grid has been changed.

English draughts is one of the many variations to the original game of draughts (Checkers). Draughts is a game that is played on a grid between two players where both players have several pieces on the grid. English draughts is played on an 8 by 8 square grid where the colours of each cell on the grid alternate. At the start of the game each player has 12 pieces. Since the size of the grid is 8 by 8 there are 64 total squares (or cells as they will be most commonly referred to in this report) however only 32 of these squares are used to play the game on since game pieces are only able to move diagonally.



The image on the right presents how a game of English draughts would be set up at the very start of the game. The image also more easily conveys the idea that only half of the cells on the grid are used to play on (only the red cells can contain a piece since pieces are only able to move diagonally).

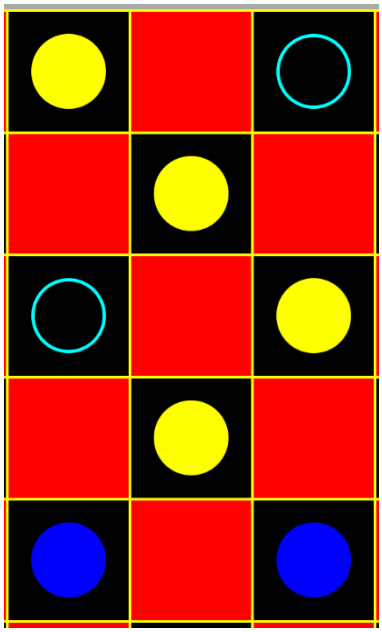
To win the game you must first remove all your opponent's pieces from the grid. To remove an opponent's piece, you need to perform a 'jump' over an opponent's piece using one of your own pieces.



Again, this idea of jumping over an opponent's piece is more easily conveyed in the second image on the right. As you can see there are two yellow pieces and one blue piece. Both of these yellow pieces are in a position where they would be able to jump over the lone blue piece and remove it from play. This is only possible because each piece has an open cell to jump to.

For example, if the cell on the top right had another blue piece within it then the left yellow piece would not be able to perform a jump. This would be the same for the right yellow piece if there was another blue piece in the top left cell.

Jumps are also able to be chained together within one move in English draughts. For example, if a piece makes a jump and then that piece is also able to make another jump from its new location then that jump Is also able to be executed within the same turn.



The image on the right shows how this is possible. The bottom right blue piece in the image can make a double jump by chaining two jumps together. Note in the image blue rings are being used to mark where this piece can move. As shown in the image the piece can make a move to the top left cell by first taking the yellow piece up and to the left. It can then immediately make another jump to the top right of it thus taking two pieces in the same move.

Another key aspect of the game is the forcing of jump moves. This essentially means that if a player has a jump move available to them on that turn then they are forced to make that jump even if it disadvantages them. If more then one jump move is available to the player, then they are able to choose which jump move to make.

2.1. Existing Solutions.

There are many online websites that offer a playable solitaire and multiplayer version of the game of checkers. The two most popular websites that I will be comparing my solution to are 247Checkers.com (Checkers, n.d.) and cardgames.io/checkers (Egilsson, n.d.).

The cardgames.io website has a lot more features when compared 247Checkers.com as well as my solution. For one the cardgames.io implementation completely forgoes the use of a ‘room code’ for connecting distributed users into a match. Instead the approach that they use involves the user entering a username into the website. This username then goes into a publicly available list on the website which when clicked on connects that user to the username they selected. The username only stays in the list while the user that entered that username is still connected to the website. While this website does feature a solitaire version of the game it doesn’t however feature any different difficulties.

The 247Checkers.com website is a lot more slimmed down then the cardgames.io website. For one the website doesn’t not feature a multiplayer version and only allows for a match to be played against the computer. The solitaire version however is a lot more advanced then its cardgames.io counterpart. It features 4 different difficulties of

'Easy', 'Medium', 'Hard' and 'Expert'. While there is a noticeable difference between the easy and expert difficulties it is hard to find a distinctive difference between for example medium and hard.

The solution I have is essentially the multiplayer aspect of the cardgames.io website combined with the more advanced solitaire version of the 247Checkers.com website (with different difficulties). Evidently the full solution I have developed is not the exact same as the solutions found on these websites. For one instead of using a username solution for connecting two users I have instead opted to user a 'room code'. A user can host a room (essentially a private game room) which has a room code. They can then share this room code with another user and then can enter it into the website. Once they do this, they will both be within the same room and the game can begin. For the solitaire version I have also chosen to only implement three difficulties instead of five as I feel that five is too many for the difficulties to be distinctively different in terms of play.

2.2. Research into Existing Approaches.

I did a lot of research into existing approaches for implementing different aspects of the game of checkers. Some of this research was done before actually starting to develop the game but most of this research was done during development. This was due to me personally wanting to write a lot of the code myself (e.g. coming up with ideas for how to handle certain aspects of the game totally by myself). This didn't end up making a lot of sense in the long run as it meant a lot of the time was wasted on code that didn't make it into the final version. This did however force me to properly research how other people had implemented certain aspects of the game and how to use certain features of JavaScript and HTML.

One of the first things I researched was HTML DOM elements and how to use them in JavaScript. I had dealt with using these elements before but had never researched how these elements can be manipulated from within JavaScript code. This research allowed me to realise that I can store these DOM elements in the JavaScript code itself. This made it much easier to move around and manipulate the DOM elements that I used to make up the pieces on the board. Storing the DOM elements in the code allowed me to more easily refer to the them as I no longer needed to refer to an element by its HTML ID (e.g. by using document.getElementById() in JavaScript).

2.3. Main Requirements for the Project.

There are two main requirements for the project, these being the complete solitaire version of the game and the complete multiplayer version of the game (from distributed machines). While these are base requirements for the project there are also other smaller included requirements. One of these is that the game works, and functions as intended. For example, both versions of the game should allow for jumps and the chaining of these jumps (as was explained in section 2). Both versions should also have a defined end point for the game (e.g. when one player has no pieces left the game should end and declare the winner of the game).

3. Data Required.

There was no human data required for this project. The only data that is requested by the user is a username. This username is used in the multiplayer version of the game, so their opponent has something to call the player they are playing against. This username doesn't have to be the users name. It can be anything they like.

4. Project Design.

Within this section of the report I will be talking about the design of the project. I will be talking about the anticipated components of the project and how they will be organised to make up the final versions of the software developed. I will also be talking about the data structures that are used throughout the developed code (such as how certain objects are used and the variables within the objects). The main algorithms used within the code will also be described (such as their uses and what function they serve). The last thing I will talk about in this section is the intended interface design. Here I will show what the interface of the developed software was intended to look like.

4.1. Anticipated Components.

As mentioned at the start of this report there will be two main components (or versions as I have referred to them) of this project. These being the multiplayer component of the game and the solitaire (single player) component. Within these components there are also smaller components that serve some functionality in making the base component work as intended. This is more evident in the multiplayer version of the game as there is a server component and a client component whereas in the solitaire version there are only the different code functions that can serve as different components. Below I will be talking about the two different components of the multiplayer version of the game.

The multiplayer version of the game utilises two main components in order to work as intended and provide a stable multiplayer environment for two people to play a game of checkers. These components are a server component and a client component. The server component does pretty much exactly what you would expect a server component to do. It acts as the middle man for all connected clients. It handles all incoming connections and handles all data that is sent from the clients to the server and handles sending necessary data out to connected clients. The server component is also responsible for handling all games of checkers currently running on the server. This involves storing the game grid for all games and handling all other necessary data (such as whose turn it currently is in the game, which clients are playing in the game and how many pieces each player has left on the grid).

The client component is the webpage that is served out to all clients when they connect to the server via a web browser. This webpage contains JavaScript code that contains functions that allow for data to be sent back to the server. When a user loads the webpage the first thing that It will do is ask the user for a username via a prompt. When this username is entered it will be sent back to the server and that username will be tied to the socket that Is currently opened between the webpage and the server. The client

component also handles the rendering of the game grid on the webpage. The grid is rendered using a standard HTML table as this is the simplest way of display a grid on a webpage and works perfectly for grid-based games. When a user wants to make a move of the grid they first need to click (or touch on touchscreen devices) on one of their pieces. Once they do this the client webpage will check with the server that it is the current players turn. If it isn't then nothing will happen. If it is then the webpage will allow a move to be made. Once a move is selected using the selection helps that are placed on the grid that move is sent to the server. The move will then be validated by the server by seeing if that move is available to be made on the server's stored version of the grid. This makes sure that any possibility of cheating is disallowed as clients do not have access to the version of the grid stored on the server.

4.2. Data Structures.

There are many different data structures that are used throughout the code of this project. Many of these data structures are arrays. These arrays are filled with objects that have specific defined inner variables which are specific to the type of the object being held. Both the multiplayer version and solitaire version of the game use these types of data structures.

4.2.1. Data Structures in the Multiplayer Version.

For the multiplayer version I will only be talking about data structures within the server component. This is mainly due to the main data structures all being within the server component as the client component only really deals with the data sent to it from the server. There are two main arrays that are used within the server component of the multiplayer version. These are the 'clients' and 'rooms' arrays. The clients array contains information on all currently connected clients such as their desired username, IP address and what room they are currently in (this will be null if they are not connected to a room). This information is stored in the client's array though the use of objects. For example, when a client first connects to the server an object for that specific client will be created. The information that was sent to the server when the client first connected will then be put into the object. This object will then be pushed into the client's array so that it can be referred to later (e.g. to look up the name of a currently connected client for example).

The rooms array contains all information related to all currently created rooms. Each room is basically a game of checkers that is currently going on between two currently connected clients. These are called rooms as a game of checkers doesn't have to be currently being played for a room to be created (for example a room is first created by just one client and a game of checkers cannot start until there are two players). Inside the rooms array are objects that made up each existing room. These objects all have

defined inner variables that are used to store information about the room. There variables are:

- Room Code. (A random four-character code of the set A-Z and 0-9 that will be used as the identifier for the room).
- The username of the first player (host of the room).
- The username of the second player.
- The grid for the room. (A 2-Dimensional array making up the grid of the game).
- The pieces on the grid.
- The username of the user whose turn it currently is in the game.
- A Boolean variable for if the game has started or not (If the first move has been made essentially).
- A Boolean variable for whether of not jumps are being forced in this game.
- The number of pieces left for the first player.
- The number of pieces left for the second player.

Once the game finishes (or both players leave the room) the room object for that room will be deleted (spliced being the technical word for this) from the array.

4.2.2. Data Structures in the Solitaire Version.

The solitaire version of the game also utilises many data structures to properly organise the data used for the game. Some examples of these data structures are the arrays used to hold the pieces of the computer player (AI) and the human player. These arrays are made up of objects where each object represents a piece that is located on the grid. These piece objects are also located on the main grid array, but they are also located inside their own separate arrays in order to find them (or more so iterate over them) more easily. Each piece object is made up of 4 variables, these being:

- Vertical position of the piece on the grid.
- Horizontal position of the piece on the grid.
- The HTML DOM element of the piece.
- A Boolean variable which represents whether the piece is a king.

Above I have mentioned another data structure that is called ‘grid’. As the name suggests this is the main grid of the game. I have left this for the end of this section as the grid array is included in both versions of the game and is the same in terms of its structure in both versions. The grid array is a 2-Dimensional array that is used to keep track of exactly where each piece is on the grid of the game. This array is needed as the grid that is used to actually display the pieces (The HTML Table) is only able to store the DOM elements of the pieces so that they can be displayed. The grid array stores the objects for those pieces which contains more important information such as its vertical and horizonal position on the grid as well as whether the piece is a king.

4.3. Algorithms.

There are many algorithms that are used throughout the code of both versions of the project. These algorithms are very important to the project as they allow the game to function correctly. Firstly, I will be talking about the algorithms present specifically in the multiplayer version of the game. I will then go on to talk about the algorithms present specifically in the solitaire version of the game. Finally, I will talk about the algorithms that are included in both the multiplayer version of the game and the solitaire version and any small differences of these algorithms (If any exist).

4.3.1. Algorithms in the Multiplayer Version.

There are two main algorithms that are used throughout the multiplayer version of the game. These algorithms are called 'validateMove' and 'createRoom'. The names of these algorithms basically describe exactly what they are used for. 'validateMove' is used for validating moves that are sent to the server by a user while 'createRoom' is used to create game rooms when requested.

The move validation algorithm works by first receiving some data from a user. This data is:

- The original vertical position of the piece to validate on the grid.
- The original horizonal position of the piece to validate on the grid.
- The vertical position that the piece wants to move to on the grid.
- The horizontal position that the piece wants to move to on the grid.
- A Boolean value represents whether the piece made a jump for this move.
- The vertical route taken.
- The horizonal route taken.

The validation algorithm will then use the original vertical and horizonal positions in another function called 'populateValidMoves'. This function will then populate an array with all the possible moves that the piece located at the given vertical and horizonal position can make. Crucially it will do this using the server's version of the grid. This ensures that cheating is impossible as the user is not able to tamper with the server's version of the grid. The validation algorithm will then iterate over the populated available moves array and will check to see if the move requested is within the array. If the move is present in the array, then the move is marked as valid and is carried out on the server's version of the grid. That version of the grid is then sent out to both players and updated on their webpages. If the move is not present in the array, then the move is marked as invalid and the player that requested that move will be told to stop tampering with the grid.

The room creation algorithm is used by the server to create rooms when requested by clients. A client can request to host a room by clicking the 'host game' button presented to

them on the webpage. Once they click this button the server will be requested to create a room and the ‘createRoom’ algorithm will be executed.

The first thing that happens when this algorithm is executed is that it will check to see if the current client is already in a room. If the client is currently in a room, then a message will be sent back to the client saying that they cannot create another room while they are currently in a room and that they will need to first leave their currently joined room (the client can do this easily by clicking the ‘leave game’ button on the webpage). Once this has been checked the algorithm will continue.

The next thing that will happen is that a random 4-character code will be created for the room. This code will be created from the set of A-Z and 0-9 and will act as the name for the room.

The algorithm will then generate the starting grid array for the room. It will put all the pieces for each player in the correct place. The player that first creates the room will ask as the ‘host’ of the room (or player 1). This player will start on the bottom half of the grid and will always take the first move.

Finally, the object for the room will be generated and the necessary information will be added into the object (such as the generated room code and grid). This object will then be pushed into the rooms array.

4.3.2. Algorithms in the solitaire version.

There is one main algorithm that is used within the solitaire version of the game. This algorithm is called ‘simComputerRound’. As is the theme with the names for these algorithms this algorithm does exactly what you would expect it to do. It simulates a round for the computer player. To do this the first thing it does is create an empty array called ‘moveablePieces’. This array will be used to store every piece that is available to be moved by the computer as well as the moves each of these pieces can make (For example for if a piece has multiple moves available to it).

The next thing the algorithm will do is populate this array. This is done by first iterating over every computer piece on the grid. A function called ‘getMoves’ will then be called on every piece. This function returns an array which is composed of all the available moves of the piece that was passed into it. This array along with the piece itself is then put into an object and pushed into the ‘moveablePieces’ array. Depending on the difficulty selected at the start of the game a move will then be selected from this array. This move will then be carried out and the algorithm will finish, and the turn will return to the human player.

4.3.3. Algorithms used in both versions.

There is one main algorithm that is used both throughout the multiplayer version of the game and the solitaire version of the game. This algorithm is called ‘placeSelectionHelpers’. This algorithm is basically used for placing the blue selection rings that appear when a human player clicks on a piece that has available moves. These rings indicate where the selected piece can move to. This algorithm is used within both versions of the game as both versions have a human player that interacts with the game. This algorithm ended up being quite complicated in how it functions. This is mainly due to it being a recursive algorithm meaning that it calls itself from within its own code.

The fact that jumps can be chained together in checkers is the reason that this algorithm needed to be recursive as the algorithm needs to be ran twice for any jump moves it comes across (Once on the starting piece and then again for the location that the piece can move to).

4.4. Intended Interface Design.

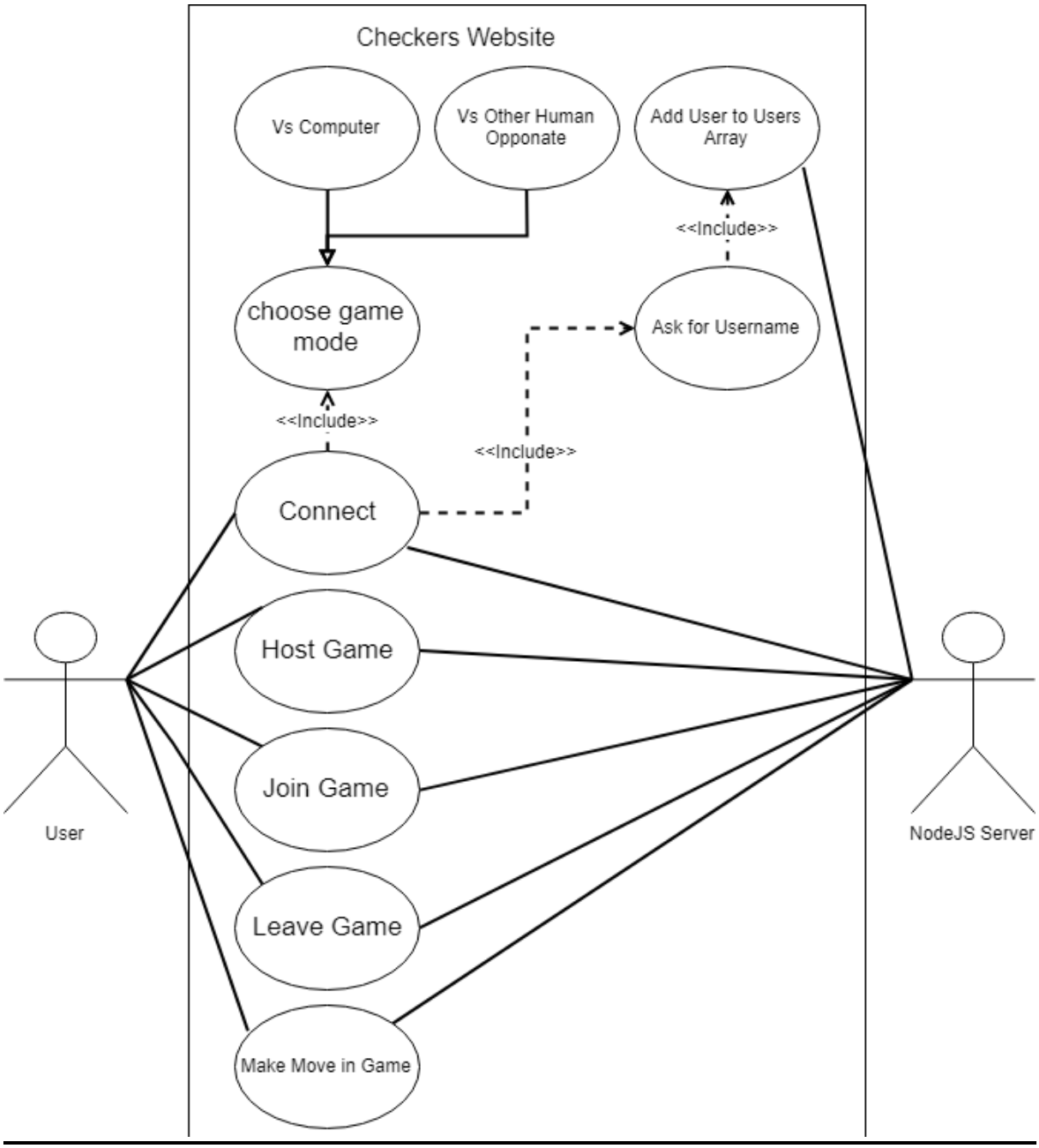
Since there are two versions of the project there will be two intended interfaces. One for the multiplayer version of the game and one for the solitaire version. These interfaces have not changed from the original interfaces in the design and specification document.

The design for the multiplayer version of the game will consist of a single webpage that is split up into 3 sections. These sections will consist of a ‘Information’ section, ‘Game Grid’ section and a ‘Chat Room’ section.

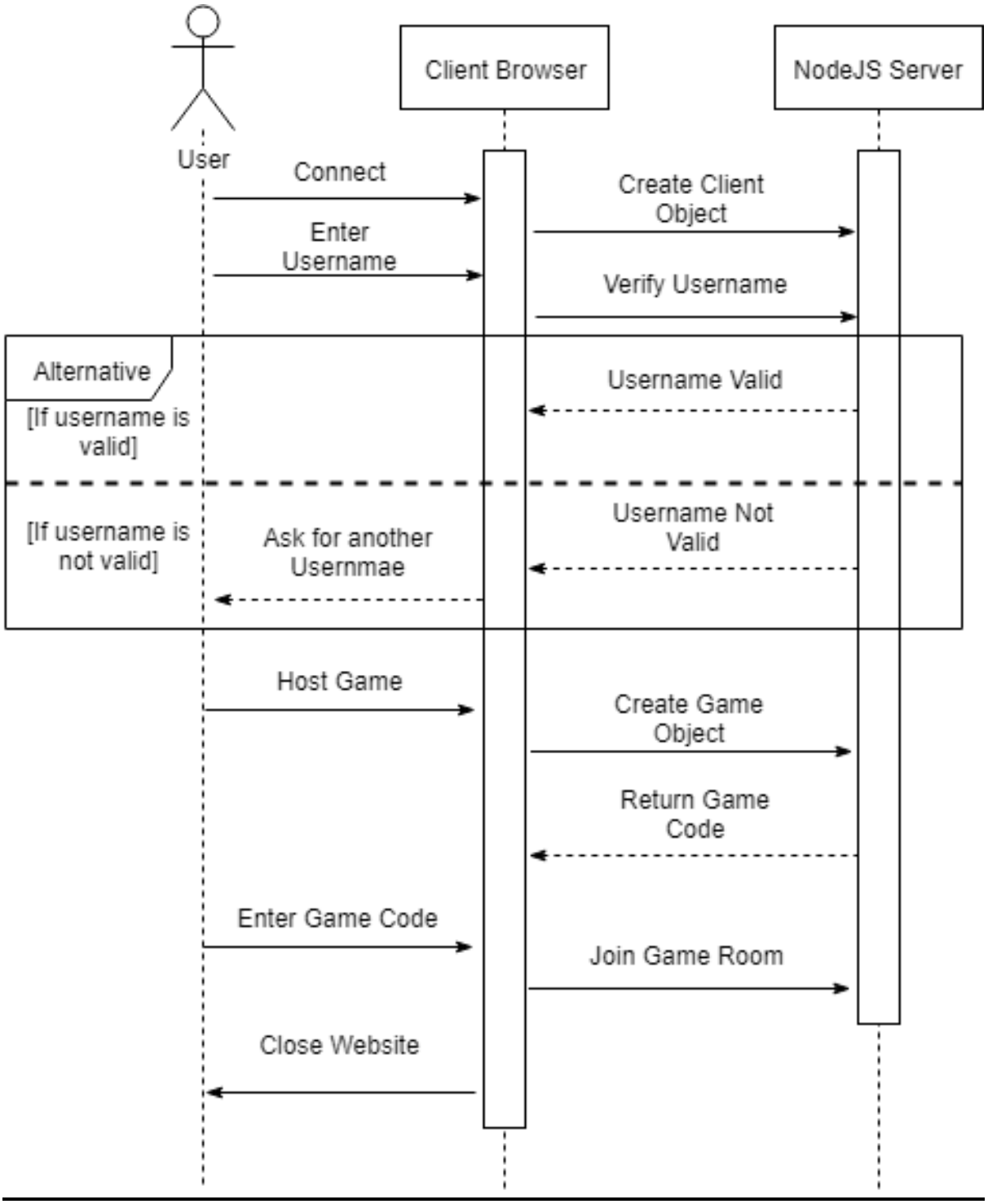
If there is enough horizonal space on the screen being used, then these sections will appear next to each other horizontally. However, If there is not enough horizonal space (for example on a smartphone) then these sections will appear on top of each other.

In the solitaire version there will be just one section that will contain the main grid for the game to be played on.

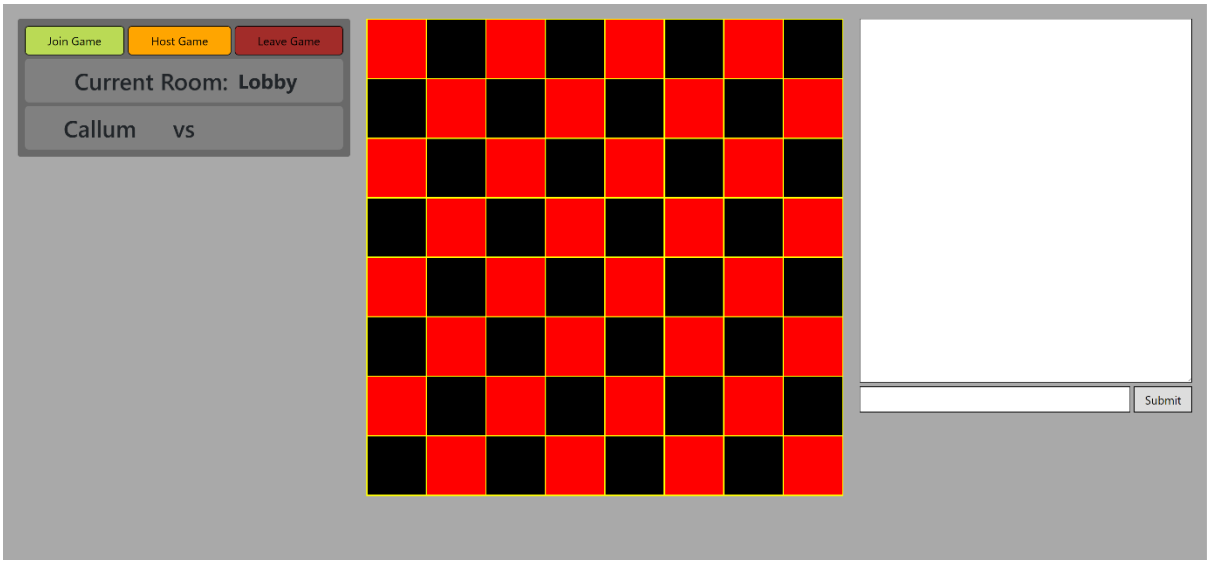
Use Case Diagram:



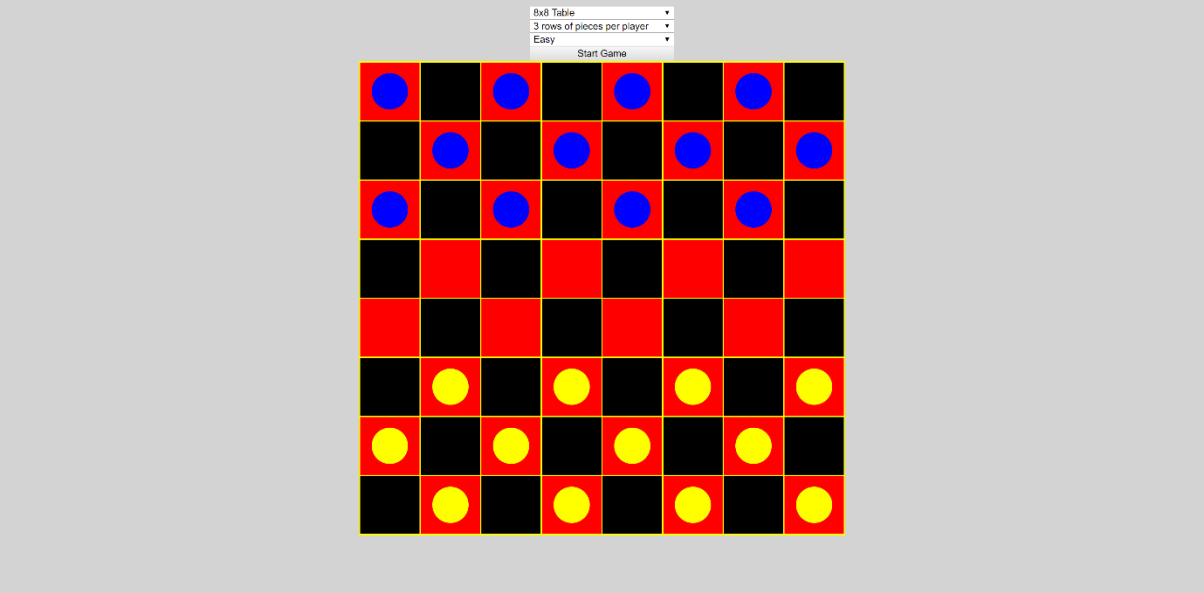
Interaction Diagram (Event Trace):



Multiplayer version Final Design:



Single Player Version Final Design:



5. Realisation.

The development of the code for this project was done in many stages. Since the project consisted of two main versions a lot of the work for these versions was split up into stages. The first version that was worked on was the multiplayer version of the game. I decided to first work on this version of the game as it included many things that I had not dealt with before (such as networking). This meant that I could find and deal with any problems I may find with developing this version early on. I then went on to work on the single player version of the game. Since these different versions are just versions of the same game a lot of the code that was used within the multiplayer version could be reused in the single player version.

5.1. Problems Encountered.

Throughout the project there were a few problems that were encountered. These problems where encountered throughout the deployment of both versions of the game. The main problem that was encountered during the development of the multiplayer version was in relation to the tracking of the movement of a piece during the chaining of jumps. Since the algorithm that placed the selection helpers for the user was located on the client side of the code It was necessary that the information about the move requested was sent back to the server so that it could be validated. If the move requested involved chaining multiple jumps together then the server needed to know the route the piece took in order to properly remove the correct piece after each jump. This required me to change the algorithm on the client side and provide two arrays that would track the route that the piece took.

The main problem that was encountered during the development of the single player version of the game was in relation to how I had planned to get the different difficulties working. Originally, I had planned to use an algorithm called MiniMax in order to decide the moves of the computer played. The way MiniMax works is it looks though all the possible moves that the computer can make and then looks at all the possible moves that could be made off those moves. This essentially means It will generate a tree of all the moves possible up to a certain depth. This was how I wanted it to work. However the MiniMax algorithm proved difficult to implement into the version of the game that I currently had and because of this I decided to not use the minimax algorithms and instead develop my own difficulties.

5.2. Testing.

There were several tests that were carried out on both versions of the game. These tests ranged from making sure that the game functioned to making sure that that certain aspects of the game worked as intended. The tests will be split up for both versions of the game.

5.2.1. Testing the Multiplayer Version.

Test	Expected Result	Actual Result
Loading the webpage	Webpage should load without any issues and should ask for a username. All sections should then be displayed to the user.	Webpage loaded without any issues and asked for a username. Once entered the webpage loaded as expected with all sections.
Hosting a game.	On clicking the 'host game' button the grid should be populated, and the room code should be shown to the user.	Grid was populated and the room code was shown to the user.
Joining a game.	On clicking the 'join game' button the user should be prompted to enter a room code. If the room code entered exists, then they should join the room and the grid should be populated and the user should be able to see the username of their opponent.	Webpage functioned as expected.
Leaving a game.	For this test the user should already be in a game. On clicking the 'leave game' button the grid should be empty, and they should be back into the lobby.	Grid was cleared and the user was put back into the lobby.
Moving a piece (when it is the users turn.)	For this test the user needs to already be in a game with another user and the turn needs to be for the current player. On clicking one of the	Webpage functioned as expected.

	users pieces the user should be presented with all the possible moves that the piece can make represented by selection helpers. On clicking a selection helper, the piece should move to that position on the grid. (grid should also update for their opponent)	
Performing a jump move	For this test the user should attempt to make a jump move over one of their opponents' pieces. This should result in the opponents piece disappearing at their piece moving to the correct location.	Webpage functioned as expected.

5.2.2. Testing the Single Player Version.

Test	Expected Result	Actual Result
Loading the webpage	Webpage should load without any issues	Webpage functioned as expected.
Making a move.	On clicking one of the user's pieces selection helpers should be placed. On clicking one of the placed selection helpers the piece should move to that location.	Webpage functioned as expected.
Testing that the computer makes a move.	When the user makes a move, the computer should make a move 3 seconds later.	Webpage functioned as expected.

5.3. Code Snapshots.

5.3.1. Code Snapshots of the Multiplayer Version.

Code for the creation of rooms:

```
socket.on('createRoom', function() { // This will be exeucted when a client attempts so host a game (e.g. create a new room).

    var clientIndex = connectedClients.indexOf(socket.id); // The index of the client in the connectedClients array.

    if (clients[clientIndex].inRoom != null) {
        socket.emit('notify', { // Tell the client the room is full.
            msg: "You are already in a room! Please leave your current room inorder to create a room.",
            alert: true
        });
        return
    }

    var allowedChars = "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"; // The set of characters that will be used to create the random 4 char code for the room.
    var generatedCode = ""; // Initializing a string var for the generated code.

    for (i = 0; i < 4; i++) { // Simple for loop which will be executed 4 times.
        var ranNum = Math.floor(Math.random() * allowedChars.length); // Select a random char from the code set.
        generatedCode += allowedChars.charAt(ranNum); // Add the selected code to the end of the genrerated code string.
    }

    var linesOfPiecesPerPlayer = 3; // Initial value for the ammount of lines of peices each player gets.

    grid = []; // Initialize grid array for the room.
    pieces = [];
```

```
for (i = 0; i < gridSize; i++) { // Create and populate the grid.
    grid[i] = [];
    for (j = 0; j < gridSize; j++) {

        grid[i][j] = {
            VPos: i,
            HPos: j,
            Type : null
        };

        if (i % 2 == 0) {
            if (j % 2 == 0) {
                //fix later
            } else {
                if (i < linesOfPiecesPerPlayer) {
                    grid[i][j].Type = "player2";
                    pieces.push({
                        VPos: i,
                        HPos: j,
                        Type: "player2",
                        isKing: startAsKings
                    })
                } else if (i >= gridSize - linesOfPiecesPerPlayer) {
                    grid[i][j].Type = "player1";
                    pieces.push({
                        VPos: i,
                        HPos: j,
                        Type: "player1",
                        isKing: startAsKings
                    })
                }
            }
        } else {
            if (j % 2 != 0) {
                //fix later
            } else {
                if (i < linesOfPiecesPerPlayer) {
                    grid[i][j].Type = "player2";
                    pieces.push({
                        VPos: i,
                        HPos: j,
                        Type: "player2",
                        isKing: startAsKings
                    })
                } else if (i >= gridSize - linesOfPiecesPerPlayer) {
                    grid[i][j].Type = "player1";
                    pieces.push({
                        VPos: i,
                        HPos: j,
                        Type: "player1",
                        isKing: startAsKings
                    })
                }
            }
        }
    }
`
```

```
    var room = {
        code: generatedCode,
        player1: client.name,
        player2: null,
        grid: grid,
        pieces: pieces,
        turn: client.name,
        gameStarted: false,
        forceJump: false,
        player1PiecesLeft: 12,
        player2PiecesLeft: 12

    }

    printWithTimeStamp("Client '" + client.name + "' created room '" + generatedCode + "'.");

    rooms.push(room); // Push the newly created room into the rooms array.
    clients[clientIndex].inRoom = generatedCode; // Update the inRoom var for the client.
    socket.join(generatedCode); // Join the socket of the client to the room.

    socket.emit('setRoom', { // Set the room on the client side.
        room: generatedCode,
        opponentName: null,
        isHost: true
    });

    socket.emit('updateGrid', { // Send the game grid over to the client for rendering.
        pieces: pieces,
        player: client.name,
        firstLoad: true
    });
})
```

Code for the validation of moves requested by a client:

```
socket.on('validateMove', function(data) {

    var clientIndex = connectedClients.indexOf(socket.id);
    var name = clients[clientIndex].name;
    var room = clients[clientIndex].inRoom;
    var roomGrid;
    var roomIndex;
    var isRoomHost;
    var playerType;
    var opponentType;
    var forceJump;
    var isKing;

    for (i = 0; i < rooms.length; i++) {
        if (rooms[i].code == room) {
            roomIndex = i;
            forceJump = rooms[i].forceJump;
            break;
        }
    }

    if (rooms[roomIndex].player1 == name) {
        isRoomHost = true;
    } else {
        isRoomHost = false;
    }

    for (i = 0; i < rooms[roomIndex].pieces.length; i++) {
        if ((rooms[roomIndex].pieces[i].VPos == data.originVPos) && (rooms[roomIndex].pieces[i].HPos == data.originHPos)) {
            isKing = rooms[roomIndex].pieces[i].isKing;
            break;
        }
    }

    var vertDir; // The vertical Direction in which the current player is playing represented by an interger. E.g. +1 is top down and -1 is bottom up.

    if (isRoomHost) {
        playerType = "player1";
        opponentType = "player2";
        vertDir = -1; // set the correct values.
    } else {
        playerType = "player2";
        opponentType = "player1";
        vertDir = 1; // set the correct values.
    }

    roomGrid = rooms[roomIndex].grid;
```

```
if (!(rooms[roomIndex].player1 == name) && !(rooms[roomIndex].player2 == name)) {
    socket.emit('notify', {
        msg: "ERROR: Player name does not exist in room",
        alert: true
    })
    return
} else if (rooms[roomIndex].turn != name) {
    socket.emit('notify', {
        msg: "ERROR: It is not the current players turn. Stop trying to cheat :)",
        alert: true
    })
    return
}

var validMoves = [];
var piecesToRemove = [];

var leapExists = checkIfLeapExists(data.originVPos, data.originHPos, isKing);

if (forceJump && leapExists) {
    populateValidMoves(data.originVPos, data.originHPos, data.originVPos, data.originHPos, data.leap, data.VRoute, data.HRoute, isKing);
} else {
    populateValidMoves(data.originVPos, data.originHPos, data.originVPos, data.originHPos, !data.leap, data.VRoute, data.HRoute, isKing);
}

if (validMoves.length == 0) {
    console.log("NON VALID MOVE");
    return;
} else {
    for (i = 0; i < validMoves.length; i++) {
        if ((validMoves[i].VPos == data.moveToVPos) && (validMoves[i].HPos == data.moveToHPos)) {
            console.log("VALID MOVE");
            rooms[roomIndex].grid[data.originVPos][data.originHPos].Type = null;
            rooms[roomIndex].grid[data.moveToVPos][data.moveToHPos].Type = playerType;
            for (j = 0; j < piecesToRemove.length; j++) {
                for (k = 0; k < rooms[roomIndex].pieces.length; k++) {
                    if ((rooms[roomIndex].pieces[k].VPos == piecesToRemove[j].VPos) && (rooms[roomIndex].pieces[k].HPos == piecesToRemove[j].HPos)) {
                        rooms[roomIndex].pieces.splice(k, 1);
                        if (isRoomHost) {
                            rooms[roomIndex].player2PiecesLeft = rooms[roomIndex].player2PiecesLeft - 1;
                        } else {
                            rooms[roomIndex].player1PiecesLeft = rooms[roomIndex].player1PiecesLeft - 1;
                        }
                        rooms[roomIndex].grid[piecesToRemove[j].VPos][piecesToRemove[j].HPos].Type = null;
                    }
                }
            }
            for (j = 0; j < rooms[roomIndex].pieces.length; j++) {
                if ((rooms[roomIndex].pieces[j].VPos == data.originVPos) && (rooms[roomIndex].pieces[j].HPos == data.originHPos)) {
                    rooms[roomIndex].pieces.splice(j, 1);
                    break;
                }
            }
```

```
                if (playerType == "player1" && data.moveToVPos == 0) {
                    isKing = true;
                } else if (playerType == "player2" && data.moveToVPos == 7) {
                    isKing = true;
                }

                rooms[roomIndex].pieces.push({
                    VPos: data.moveToVPos,
                    HPos: data.moveToHPos,
                    Type: playerType,
                    isKing: isKing
                })

                var opponentName;

                if (playerType == "player1") {
                    opponentName = rooms[roomIndex].player2;
                } else {
                    opponentName = rooms[roomIndex].player1;
                }
                rooms[roomIndex].turn = opponentName;
                rooms[roomIndex].gameStarted = true;
                io.sockets.in(room).emit('updateGrid', {
                    pieces: rooms[roomIndex].pieces,
                    player: opponentName,
                    firstLoad: false
                })
                console.log("player 1 has " + rooms[roomIndex].player1PiecesLeft + " pieces left.");
                console.log("player 2 has " + rooms[roomIndex].player2PiecesLeft + " pieces left.");

                if (rooms[roomIndex].player1PiecesLeft == 0) {
                    io.sockets.in(room).emit('notify', {
                        msg: rooms[roomIndex].player2 + " has won the game!",
                        alert: true
                    })
                } else if (rooms[roomIndex].player2PiecesLeft == 0) {
                    io.sockets.in(room).emit('notify', {
                        msg: rooms[roomIndex].player1 + " has won the game!",
                        alert: true
                    })
                }
                return;
            }
        }
        console.log("NON VALID MOVE");
    }
```

5.3.2. Code Snapshots of the Single Player Version.

Code for returning all the possible moves of a piece:

```
function getMoves(passedGrid, VPos, HPos, checkClose) {

    var avaliableMoves = [];

    var opponentPieceType;
    var vertDir;

    if (passedGrid[VPos][HPos].Type == "player") {
        vertDir = -1;
        opponentPieceType = "computer";
    } else {
        vertDir = 1;
        opponentPieceType = "player";
    }

    if (!(VPos + vertDir < 0) && !(HPos - 1 < 0) && checkClose && !(VPos + vertDir >= grid.length)) { // Check close Left;
        if (passedGrid[VPos + vertDir][HPos - 1].Type == null) {
            avaliableMoves.push({jump: false, moveToVPos: VPos + vertDir, moveToHPos: HPos - 1, removeVPos: null, removeHPos: null});
        }
    }
    if (!(VPos + (vertDir * 2) < 0) && !(HPos - 2 < 0) && !(VPos + (vertDir * 2) >= grid.length)) { // Check leap Left;
        if (passedGrid[VPos + (vertDir * 2)][HPos - 2].Type == null && passedGrid[VPos + vertDir][HPos - 1].Type == opponentPieceType) {
            avaliableMoves.push({jump: true, moveToVPos: VPos + (vertDir * 2), moveToHPos: HPos - 2, removeVPos: VPos + vertDir, removeHPos: HPos - 1});
        }
    }
    if (!(VPos + vertDir < 0) && !(HPos + 1 >= grid.length) && checkClose && !(VPos + vertDir >= grid.length)) { // Check close Left;
        if (passedGrid[VPos + vertDir][HPos + 1].Type == null) {
            avaliableMoves.push({jump: false, moveToVPos: VPos + vertDir, moveToHPos: HPos + 1, removeVPos: null, removeHPos: null});
        }
    }
    if (!(VPos + (vertDir * 2) < 0) && !(HPos + 2 >= grid.length) && !(VPos + (vertDir * 2) >= grid.length)) { // Check leap right;
        if (passedGrid[VPos + (vertDir * 2)][HPos + 2].Type == null && passedGrid[VPos + vertDir][HPos + 1].Type == opponentPieceType) {
            avaliableMoves.push({jump: true, moveToVPos: VPos + (vertDir * 2), moveToHPos: HPos + 2, removeVPos: VPos + vertDir, removeHPos: HPos + 1});
        }
    }
```

```
    if (passedGrid[VPos][HPos].isKing) {

        console.log("Piece is king");

        if (!(VPos + (vertDir / -1) < 0) && !(HPos - 1 < 0) && checkClose && !(VPos + vertDir >= grid.length)) { // Check close Left;
            if (passedGrid[VPos + (vertDir / -1)][HPos - 1].Type == null) {
                avaliableMoves.push({jump: false, moveToVPos: VPos + (vertDir / -1), moveToHPos: HPos - 1, removeVPos: null, removeHPos: null});
            }
        }
        if (!(VPos + ((vertDir / -1) * 2) < 0) && !(HPos - 2 < 0) && !(VPos + (vertDir * 2) >= grid.length)) { // Check leap Left;
            if (passedGrid[VPos + ((vertDir / -1) * 2)][HPos - 2].Type == null && passedGrid[VPos + (vertDir / -1)][HPos - 1].Type == opponentPieceType) {
                avaliableMoves.push({jump: true, moveToVPos: VPos + ((vertDir / -1) * 2), moveToHPos: HPos - 2, removeVPos: VPos + (vertDir / -1), removeHPos: HPos - 1});
            }
        }
        if (!(VPos + (vertDir / -1) < 0) && !(HPos + 1 >= grid.length) && checkClose && !(VPos + vertDir >= grid.length)) { // Check close Left;
            if (passedGrid[VPos + (vertDir / -1)][HPos + 1].Type == null) {
                avaliableMoves.push({jump: false, moveToVPos: VPos + (vertDir / -1), moveToHPos: HPos + 1, removeVPos: null, removeHPos: null});
            }
        }
        if (!(VPos + ((vertDir / -1) * 2) < 0) && !(HPos + 2 >= grid.length) && !(VPos + (vertDir * 2) >= grid.length)) { // Check leap right;
            if (passedGrid[VPos + ((vertDir / -1) * 2)][HPos + 2].Type == null && passedGrid[VPos + (vertDir / -1)][HPos + 1].Type == opponentPieceType) {
                avaliableMoves.push({jump: true, moveToVPos: VPos + ((vertDir / -1) * 2), moveToHPos: HPos + 2, removeVPos: VPos + (vertDir / -1), removeHPos: HPos + 1});
            }
        }
    }
    return avaliableMoves;
}
```

Code that returns a Boolean value for If a piece has a jump move available to it:

```
function checkIfLeapExists(isPlayer) {

    var vertDir;
    var opponentPieceType;
    var pieceArray;

    if (isPlayer) {
        vertDir = -1;
        opponentPieceType = "computer";
        pieceArray = playerPieces;
    } else {
        vertDir = 1;
        opponentPieceType = "player";
        pieceArray = computerPieces;
    }

    for (i = 0; i < pieceArray.length; i++) {

        VPos = pieceArray[i].VPos;
        HPos = pieceArray[i].HPos;

        if (!(VPos + (vertDir * 2) < 0) && !(HPos - 2 < 0) && !(VPos + (vertDir * 2) >= grid.length)) { // Check Leap Left;
            if (grid[VPos + (vertDir * 2)][HPos - 2].Type == null && grid[VPos + vertDir][HPos - 1].Type == opponentPieceType) {
                return true;
            }
        }
        if (!(VPos + (vertDir * 2) < 0) && !(HPos + 2 >= grid.length) && !(VPos + (vertDir * 2) >= grid.length)) { // Check Leap right;
            if (grid[VPos + (vertDir * 2)][HPos + 2].Type == null && grid[VPos + vertDir][HPos + 1].Type == opponentPieceType) {
                return true;
            }
        }
        if (pieceArray[i].isKing) {
            if (!(VPos + ((vertDir / -1) * 2) < 0) && !(HPos - 2 < 0) && !(VPos + (vertDir * 2) >= grid.length)) { // Check Leap Left;
                if (grid[VPos + ((vertDir / -1) * 2)][HPos - 2].Type == null && grid[VPos + (vertDir / -1)][HPos - 1].Type == opponentPieceType) {
                    return true;
                }
            }
            if (!(VPos + ((vertDir / -1) * 2) < 0) && !(HPos + 2 >= grid.length) && !(VPos + (vertDir * 2) >= grid.length)) { // Check Leap right;
                if (grid[VPos + ((vertDir / -1) * 2)][HPos + 2].Type == null && grid[VPos + (vertDir / -1)][HPos + 1].Type == opponentPieceType) {
                    return true;
                }
            }
        }
    }
    return false;
}
```

6. Evaluation.

Within this section I will be evaluating the system created as a whole. I will be evaluating both the multiplayer version and the single player version of the project. I will not however be including any third-party evaluation. In my design and specification document I did say that I would create a questionnaire to give to people to evaluate the project. I however did not think this was worth the effort of getting the questionnaire approved by the university so this part of the evaluation will not be included.

Overall, I feel the project has gone well. While I do think certain aspects of the project have gone better than others, I do think that the code developed is of a good quality. However, I do think that the development of the single player version of the game could have been better. For example, the difficulties included with the single player versions are not of the quality that I was hoping to achieve when I started the project. I am disappointed that I did not end up getting the MiniMax algorithm to work with my system. The single player version of the game would have been a lot more complex if MiniMax was being used to decide moves for the computer player.

While I might not be very happy with how the single player version of the game turned out I am very happy with the multiplayer version. The multiplayer version of the game functions exactly how it was set out to. It allows two players from distributed machines to play a game of checkers. The server component of the system allows for multiple games to be played at the same time and can handle a respectable number of players connected at one time.

6.1. Strengths of the Project.

As mentioned above I feel the main strength of the project is the multiplayer version. However, there are other strengths that are related to both versions of the project. In my opinion one of these strengths is the use of data structures such as arrays and objects that are used throughout the code. For example, the 2-Dimensional array used to keep track of the grid was very useful for both versions of the game. The objects that were used for each piece on the grid was also very useful as it allowed me to keep track of all the pieces.

Another strength of the project would be its overall design in terms of its interface. The interface for both versions Is very minimalistic and I feel is quite simple to understand.

6.2. Weaknesses of the Project.

Again, as mentioned above I feel the main weakness of the project is the single player version. This version of the did not end up being what I had originally wanted it to be. The difficulties on offer are no where near as complex as I had hoped they would have turned out. This is mainly due to me finding it quite difficult to develop these difficulties as developing this sort of code is not something that I am particularly good at. If I was to redo this project from start this is something that I would have done a lot more research into.

7. Learning Points.

This project presented me with many learning points while it was being carried. Most of these learning points came from the multiplayer version of the game. This is mainly due to me not having developed any distributed multiplayer games in the past (distributed meaning that each player is playing from a different machine and might be on a different network). In order to get the multiplayer version of the game working I had to learn a lot about how a multiplayer environment in JavaScript may be set up. This led me to two different JavaScript libraries that helped massively with the networking side of the multiplayer version of the game. These being Node.JS and Socket.io. Since I had not used either of these before I had to learn a lot about how they work in order to utilise them effectively for the project.

Node.JS is a JavaScript runtime that allows for JavaScript code to be executed outside of the standard browser environment. This essentially allows for a server to be executed which allows for JavaScript code to be executed on it. Connections from outside sources (e.g. from different networks via the internet) can be established and handled concurrently. Once I had done some research into how to use and set up a node.JS server I found it very easy to program the networking side of the multiplayer version. The Node.JS website (Node.js v12.2.0 Documentation, n.d.) helped immensely with this as the website provides a ‘Manual and documentation’ page which is filled with information on how to use and set up a Node.JS server and fully utilise all the features that it provides.

Socket.IO is a JavaScript library that is used specially with a Node.JS server. Socket.IO allows for traditional web sockets to be used within node.JS. Socket.IO is essentially the library that facilitates the sending of packets between clients connected to the server. For example, a packet of data can be sent to one individual client or multiple clients from the server. Once this packet is received by a client a specific block of code will be executed and use the data that was sent within the packet. This can also work the other way around with packets being sent from a client to the server.

8. Professional Issues.

While carrying out this project I have made sure to adhere by the codes of practice and conduct that is issued by the British Computer Society. One of the first practices mentioned in this document relates to making sure that my technical competence is being maintained. I feel that I made sure that my technical competence was being maintained while carrying out this project as this project required me to do a lot of technical programming and research.

Another practice that is mentioned in this document relates to making sure that I am adhering to regulations. I have read though these regulations and have made sure that my project doesn't not break any of these regulations.

Another practice relates to acting professionally as a specialist. I also feel that I have followed this practice as I have done nothing unprofessional while carrying out this project.

Another practice relates to using appropriate methods and tools. I feel that I have also definitely followed this practice as I used many methods and tools throughout the project. An example of this would be my use of Node.JS and Socket.IO for the networking portion of the multiplayer version of the game.

Manging my workload efficiently is another practice that is mentioned in the codes of practice document. This is something that I could have put more effort into doing as my time was not managed very well towards the end of the project.

Participating maturely is another practice that is mentioned in the codes of practice document. I feel I have also followed this practice as I have done nothing immature throughout the development of this project.

9. Bibliography.

Bibliography

247Checkers. (n.d.). Retrieved from https://www.247checkers.com

BCS. (n.d.). *Code of Good Practice*. Retrieved from https://www.bcs.org/upload/pdf/cop.pdf

Checkers. (n.d.). Retrieved from https://www.chessprogramming.org/Checkers

Egilsson, E. (n.d.). *CHECKERS*. Retrieved from https://cardgames.io/checkers/

Gambetta, G. (n.d.). *Fast-Paced Multiplayer (Part I): Client-Server Game Architecture*. Retrieved from http://www.gabrielgambetta.com/client-server-game-architecture.html

Męciński, M. (n.d.). Retrieved from Medium: https://medium.com/@MichalMecinski/architecture-of-a-node-js-multiplayer-game-a9365356cb9

Node.js v12.2.0 Documentation. (n.d.). Retrieved from https://nodejs.org/api/cluster.html

Strategy Game Programming Tutorial. (n.d.). Retrieved from http://www.fierz.ch/indepth.php

University, S. (n.d.). *ALGORITHMS - MINIMAX*. Retrieved from https://cs.stanford.edu/people/eroberts/courses/soco/projects/2003-04/intelligent-search/minimax.html

What Socket.IO is. (n.d.). Retrieved from https://socket.io/docs/