1 April 2016 CS1006 Practical 2 Student ID: 150012773

Backgammon

Overview

The second practical required us to produce an implementation of game, Backgammon, that can be played by human players and/or AI players over a network or on a local machine. We had to create an interface which would allow human players to see the see the current state of the board and thus play, or observe an AI player. The implementation we had to write will allow a single player (human or AI) to observe the board and make moves. The opponent player will be able to use a different instance of the same interface to interact with the game. In this way, we will be able to achieve a simulation of a game by running two instances of our software, one for each player, either on the same computer or different computers.

Design

UI

We decided to create a graphical user interface, using a JFrame, swing components and images. When the program is initiated, a new JFrame is created, the frame is a fixed size and cannot be resized. The background of the frame is set to a picture of an empty backgammon board. The pieces are then added and are represented by either a picture of a red circle or a white circle. Finally a “Roll Dice” button is added and labels which display whose turn it is and each player's score.

In order to keep track of each piece, we created a class named GUI pieces, each piece is then assigned the correct image, the pixel coordinates of that piece are stored and the type that piece is, is also stored e.g. red or white.

Next in order to visually implement the dice, we have simply used rectangles and ovals to represent the face of the dies. When the user rolls the dice, to give the illusion of actually rolling we have used a timer which will execute a procedure x ammount of times until the timer finishes. Each execution will change the quantity of ovals printed on each die to a random number between 1 and 6. When the time has finished the number of dots on the dice is recorded by the game engine and saved. If the user roll a double, two more dice are displayed with the same values shown. Once a player has then moved a piece the one of the die will be removed the board and until either there are no more die left (the user has played all their moves) or the user can no longer move. In both cases the game will automatically end the current users turn and start the opponents turn.

When two human user play against one another, two instances of the board are displayed. Each instance is simultaneously updates with the board moves. Each instance contains labels which signify which player you are (RED or WHITE) and whether or not it is your turn.

Finally each instance also contains labels containing both your score and your opponents score.

In order to allow the user to select a piece which they wish to move, we have implemented and added action listeners to the “BackgammonGUI” class. This allows us to detect when a mouse click has occurred on the frame where the user is not clicking a button. By using the coordinates of where the user clicked we are then able to calculate which if any counter has been clicked by finding a counter with that coordinates set. Therefore if its the users turn and they have available moves, they are able to select a counter by clicking on it, once this happens the image of the clicked on counter is changed to a green circle, signifying it has been selected. At this stage the coordinates of the counter are translated from the pixel coordinates of the frame to a new set of coordinates which range from x:0-12 and y:0-1. This is acvived by using the “CoordinatesUtil” class. The translation is necessary as it allows us to ensure the new selected location is valid and moreover complies with the dice that are rolled. So for example if a 5 is rolled the red counter selected can only move 5 spaces in the correct direction.

Now in order to move the highlighted piece the user is required to re-clicks on the board where are again the coordinates of the new click are translated. Based on where the user has clicked on the board one of several methods can be executed. These method include “moveIfValid”, “reenterIfValid” and “bearOffIfValid”. Each of these methods uses logic stored in the game engine to ensure the move is valid.
If the user the is able to move pieces and whishes to simply move a piece from one board location to another the “moveIfValid” method is called. If the user the is able to bear pieces off and clicks on the draw the “bearOffIfValid” method is executed and finally if the user has any pieces captured they will have to try and re enter before making any other moves, in this event the “reeneterIfValid” method is called.
Once a valid move has been executed the counter image is then set back to the correct colour, the frame is repainted and the on screen location of the counter will have changed to the desired position. This is achieved by using multiple lists which include a list of” “GUIPiece” objects which contains all 30 of the pieces in the game. “RedBornOff” and “WhiteBornOff” which contains the red “GUIPiece” objects and white “GUIPice” objects that have be born off respectively. The final lists include “RedCaptured” list and “WhiteCaptured” list which contains the red “GUIPiece” objects and white “GUIPice” objects that have be captured respectively.
Every modification to the board then causes the “GUIPiece”, “RedCaptured” and “WhiteCaptured” lists to be cleared and repopulated with updated pieces, finally the contents of the updated lists are then applied to the board by using the paint method which loops through all the lists and paints the correct images in the correct locations on the board.
If the one player attempts to capture an opponent's piece and is able to do so the move will be validated and executed. Then once the board is updated and repainted the captured piece will appear either at the top or the bottom of the board in the bar for red and white

pieces respectively. The captured piece can then be able to be selected from the bar when the opponent starts their turn and if they are able to reenter the piece to their start board, they can do so. In this event the captured piece will removed from the bar and reintroduced to the board at the chosen location, upon repaint.
Once a player has all their pieces in their home board they will be able to make bearing off moves, providing they have the correct dice rolls available. In this event the user must select the piece they wish to bear off, then click on their bearing off area. If the game engine validates the move, the user's piece will be added to the “RedBornOff” list if they are playing as red, or the “WhiteBornOff” list if they are playing as white. When the board is then updated and repainted the correct bear off area will then contain a new image (the side view of the counter) and the players score label will be incremented. To give the illusion of stacking the born off counters in the bear off area, every time a piece is born off a new image is added to the correct born off area, above the previous image.

To make the interface more user friendly and generally help players, we have implemented a system which calculates where a selected piece can move to. The available locations where the selected piece can move to are then highlighted green. To achieve this, when a user selects a piece a method called “showAvailableMoves” is used and the coordinates of the counter as well as the counter colour is passed as parameter.
From here the game engine is used to calculate available moves from a set position given the colour of a counter and a list containing instances of the interface Move (implementations of the Move interface can include: “BoardMove”, “ReenterMove”, “BearOffMove” as well as “NullMove” and “InvalidMoveException” although these to last implementations are not used by the by here) is generated. The generated list is then iterated through, and the program determines which class the current Move belongs too and the end location of the move, each end location is then added to a list “highlightedCounterPos”. When the board is updated and repainted a new green triangle or square, for board/re-enter moves or bear off moves respectively will be added to the locations where the selected counter can be moved to.

Finally to prevent large stacks of counters building up the UI uses labels to signify the number of counters at a given location. The labels are used to display the number of counters that are stacked on top of one another. Once 6 counters are being displayed in the same column, any additional counters moved to this column are not displayed, but instead a label is show to represent how many counters additional counters are in the given column. This is also happend where pieces are born off. To achieve this a list of coordinates is defined at the start of the program which contains the locations of where all the labels should be displayed, a list of labels is also generated at the start, where each label is empty. When the program is updating the board it will check if any column contains more than 6 counters or if the bar contains more than one counter for each player. If so a label is displayed which signifies how many additional counters are in that location at the correct position.

AI

We have built two different AI's which use different techniques to find the best available move. The methods we have used are both based on the minimax algorithm. The first method we have used is the minimax algorithm, which makes decisions that which will minimise the possible loss for a worst case (maximum loss) scenario.

The minimax method we have implemented is a recursive method for choosing the next move. The game engine is used used to calculate all the available moves, either given a set of dice, or for all dice combinations. Each calculated game state is represented as an instance of the "TurnNode" class and a value is then associated with calculated game state. This value is computed by means of evaluation method which in the "StaticEvaluator" class. The associated score indicates how good it would be for the AI to make a specific move. The best move chosen by the AI is the move that maximizes the minimum value of the position resulting from the opponent's possible following moves.

With the use of our heuristic evaluation function which gives values to non-final game states without considering all possible following complete sequences. We have limited the minimax algorithm to look only at a certain number of moves ahead.

The alternative method we have implemented is the expectimax algorithm, which works in a similar way to the minimax method although is a more specialized variation of a minimax method. The minimax algorithm is specifically designed for play two-player zero-sum games with a random element involved which makes it perfect for backgammon as the outcome takes takes into account the probability of different dice rolls.

In addition to "min" and "max" nodes of the minimax tree, this variant has chance nodes, which take the expected value of a random event occurring.

In the minimax method, the levels of the tree alternate from max to min until the depth limit of the tree has been reached. In an expectiminimax tree, the "chance" nodes are interleaved with the max and min nodes. Instead of taking the max or min of the evaluation values of their children, chance nodes take a weighted average, with the weight being the probability that that child is reached.

To avoid long computation time, and to allow our program to look a greater number of moves ahead we have implemented a technique called alpha-beta pruning. The reason we have introduced alpha-beta pruning is to decrease the number of game states that are evaluated by our minimax and expectimax method. With the use of alpha-beta pruning, both methods will stop completely evaluating a branch of possible states when at least one possibility has been found that proves the move to be worse than a previously examined move and thus such moves are not being evaluated further. The move returned is the same move as both methods previously returned, but branches that cannot possibly influence the final decision are no longer evaluated. To implement both methods now maintain two values, "alpha" and

“beta”, which represent the maximum score that the maximizing player is assured of and the minimum score that the minimizing player is assured of respectively.

Initially alpha is negative infinity and beta is positive infinity, i.e. both players start with their lowest possible score. It can happen that when choosing a certain branch of a certain node the minimum score that the minimizing player is assured of becomes less than the maximum score that the maximizing player is assured of (beta<=alpha). If this is the case, the parent node should not choose this node, because it will make the score for the parent node worse. Therefore, the other branches of the node do not have to be explored.

The benefit of alpha–beta pruning lies in the fact that branches of the search tree can be eliminated. This way, the search time can be limited to the 'more promising' subtree, and a deeper search can be performed in the same time. The optimization reduces the effective depth to slightly more than half that original methods.

The evaluation function used to assign scores to game states, takes into account multiple different conditions and assigns scores based on the weights assigned to the conditions. The conditions we are primarily focusing on include: “captureWeight”, “bearedOffWeight”, “uncoveredWeight” and “distanceToTravelIWeight”.

Logic

During the implementation if this practical, we tried as much as possible to implement good programming practices, both in writing the code and in structuring the classes. This led us into separating the game’s logic from the various players that could interface with it.

The bulk of the game logic is split between two classes: Board and Game.

Board

The Board holds most of the logic behind moving the pieces and validating said moves. The board state itself is implemented as an integer array where negative values represent the red pieces and positive values represent the positive pieces, with the absolute value of this representing the number of pieces on a point.

In addition of that, separate variables are used for holding the number of pieces born off and captured for each player.

We’ve chosen to use integers instead of some specialized object for each of the points as it facilitates both serializing the object and cloning it which is helpful for generating the AI search tree.

1 April 2016 CS1006 Practical 2 Student ID: 150012773

1 April 2016 CS1006 Practical 2 Student ID: 150012773

IPlayer

This interface is implemented by all the classes that act as players in this project. This defines a series of events that the player can receive, and it is the player's duty to respond to these events in a proper manner.

Game

The game class contains the logic for switching the turns between players and acts as a proxy between the board and the players.
The player movement is bound in both classes to their pieces' colors, red moving from 0 (which is in the bottom right corner) to 24 (which is in the top right corner), and white the other way around, from 24 to 0.

Moves

To improve the ease in creating both the AI and the networked communication, moves have received their on class. The moves can be applied on both games (for the networking and normal gameplay) and the boards (extensively used in the AI).

The necessity of a NullMove arised when writing the networking code, as we needed a way to represent unused moves.

Even though we represent the moves as a separate entity, they still rely on the board to validate and execute the moves.

The failed attempt at multithreading

Most of the classes are littered with synchronization attempts, as we tried to separate the GUI thread from the Game and Networking thread. Unfortunately, due to our inexperience in the subject, we encountered some race conditions which we didn't manage to solve. Due to time constraints these attempts weren't removed from the classes, as they don't affect the functionality of the game itself.

Part of this attempt are the EventHandling classes and the events package. Events were used to notify the game and the players of various things, such as yourTurn for Player, or Move for game.

We decided to keep these classes in the project both show our effort and also to allow us to maybe try to reimplement this functionality in the years to come.

Network

Even though we were given an example TCP chat client which most of the teams decided to adapt for their own uses, we decided to write our own servers and clients, as we thought that would allow us to better understand networking.

I consider our network code a success, as it completely and accurately implements the protocol presented in the specification.

The code was initially written as two separate classes, NetworkClient and NetworkServer, but later most of the common code between those two was pulled into a third abstract class called NetworkPlayer, which the two initial classes inherit from.

The parsing and generating of the turn messages is delegated to a Turn class, which is filled out by the game and by the network interfaces. This class contains the data sent through the network, which is the dices rolled and the moves executed with those dices.
This class was also used in the AI, when constructing the Turn that it should play.

Testing

Testing the game

To test the program and make sure it works as expected we will run the program as it expected to be run and play the game through to the end numerous time trying to do as many combinations of moves. In doing this all instances of all classes will be initiated. Throughout our program we have also incorporated many try and catch error statements which allow my program to catch exceptions and handle exceptions appropriately. The main validation checks we will be testing is the moving, bearing off, capturing and re entering of pieces. To ensure that the program can recognize when a user is trying to make an invalid move we have created an “InvalidMoveException” class. Instances of this class should be initiated upon invalid move attempts.
Moreover because we have used a GUI there and not a text based interface there are few time the user will have to input text to the program and thus we will not have to do to many verification and validation tests, so for example entering a number instead of a string for a move to coordinate.
However, we will, where possible, enter invalid data in an attempt to break the program, by for example making an invalid move and therefore perform both extreme and erroneous tests. If the program works as expected, the program will be able to deal with the exceptions

thrown and the user should be notified that the attempted move is invalid. Upon this happening the user will be requested to make a new move which is valid and until this is done the program will not continue.

Testing the AI

In order to test the AI, the analytics package was created. This contains a few various test, which are documented below.
The tests contain some bad games. Those are games that ended without assigning a winner. We have narrowed the problem down to in our calculation of the bearing off moves, but due to time constraints, we weren't able to fix it.

The testing was implemented by running multiple games at once with the help of a Thread Executor Pool.

Test for bias

This is a test for checking if our AI or our evaluation function is biased towards one of the counter types.
Unfortunately, our program fails this test, as it seems that our evaluation function is biased towards the red pieces. Again, due to time constraints, this bias could not be remediated.
Running 1000 games between the same AI yielded the following results:

- Red wins: 695
- White wins: 273
- Bad games: 32

The bias is off about 20% off the 50/50 required for an unbiased system.

Difference in depth test

This test is written to test if increasing the number of moves the game looks ahead improves the game's performance.
After running 20 games, this are the results:

- Red wins: 16
- White wins: 3
- Bad games: 1

Test against random AI

This test is a sanity check, to see if our AI is better than a random AI.
After running 10 games, these are the results:

- Red wins: 9
- White wins: 0
- Bad games : 1

After these tests, I consider that our AI behaves better than one that just chooses random moves, although further testing is necessary to reach a concrete decision.

Testing the network

The network was thoroughly tested throughout development. I consider that it implements the protocol specified to the letter, although it could do with some improvements in handling the errors.

The networking code was also used to successfully communicate with other player's implementation, which further increases our confidence in our application.

Core functionality tests

Description	Expected Result	Actual Result	Comment
Select a piece	The selected piece is made green	The selected piece was highlighted green	Worked as expected
Select opponent's piece	The program does not respond	None of the opponent's pieces could be selected, attempts were ignored.	Worked as expected
Roll dice when not your turn	Clicking the roll dice button will not perform any task	Nothing happened, the user could not use the roll dice button until it was their turn	Worked as expected
Roll the dice	The dice appear to roll for 100ms	The faces of the dice changed multiple time for a fixed period and displayed random numbers	Worked as expected
Move piece to valid destination	The piece will move from original location to new location	The selected piece was removed from the old location and appeared in the desired location	Worked as expected
Move piece to invalid destination	The game will show a pop up box saying the move was invalid and will not continue until a valid move is made	The game displayed a pop up box saying the move was invalid and did not continue	Worked as expected
Bear Piece Off	The piece will be removed from the board, the bear off area will contain a new image of counter and the score is incremented	The piece was removed from the board, the bear off area was updated and showed a new image of counter and the score was incremented	Worked as expected for most cases, although it causes the AI to give invalid moves sometimes.

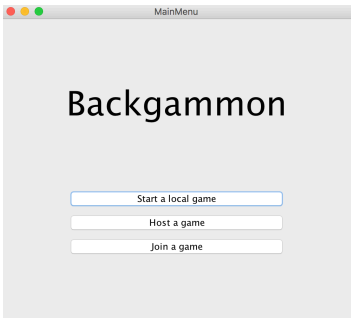
Attempt beare off when not all pieces in home board	The game will show a pop up box saying the move was invalid and will not continue until a valid move is made	The game displayed a pop up box saying the move was invalid and did not continue	Worked as expected
Capture Piece	The piece capturing opponent's piece will move to the correct location and the captured piece will be displayed on the bar in the correct location	The was correct displayed in the new location and the captured piece will was displayed on the bar in the correct location	Worked as expected
Attempt invalid capture opponent's piece when more than 1 of opponents counter in column	The game will show a pop up box saying the move was invalid and will not continue until a valid move is made	The game displayed a pop up box saying the move was invalid and did not continue	Worked as expected
Reenter piece	The piece will be removed from the bar and will appear in the desired location on the board	The piece was removed from the bar and was displayed in the desired location on the board	Worked as expected
Can't Make a move	The game automatically switches player and current player turn is ended	The game automatically switched player and current player turn was ended	Worked as expected
Can only make 1 move	The game allows the player to make the available move then automatically switches player and current player turn is ended	The game allowed the player to make an available move then automatically switched player and current player turn was ended	Worked as expected
Make 4 available moves	The player can make 4 moves	The player was allowed to make 4 moves	Worked as expected
Winning the game	The game is terminated and the displayers which	The game terminated and displayed which	Worked as expected

	player won	player won in the console	
Losing the game	The game is terminated and the displayers which player won	The game terminated and displayed which player won in the console	Worked as expected
Quitting the game	The game displays a message in the console notifying the other player that oppolent quit	The game displayed a message in the console notifying the other player that oppolent quit	Worked as expected
Ensure highlighted location correct	The program will highlight all the correct locations that the selected counter can move too	The program did highlight all the correct locations that the selected counter can move too	Worked as expected
Ensure Scores correct	The score are correctly incremented and both instances of the game show the same scores	The score were correctly incremented and both instances of the game show the same scores	Worked as expected
Capture multiple pieces	After a player has more than one piece on the bar, no more pieces should be displayed on the bar but a label should be added signifying how many pieces that user has captured. When pieces were re entered the number on the label should update, until no pieces on the bar and the label should removed	After a player had more than one piece on the bar, no more pieces were displayed on the bar but a label was added signifying how many pieces that user had captured. When pieces were re entered the number on the label correctly updated, until no pieces were in the bar and the label was removed	Worked as expected
More than 6 counters in column	After a player has more than six piece in the same column, no more pieces should be displayed	After a player had more than six piece in the same column, no more pieces were be displayed on that	Worked as expected

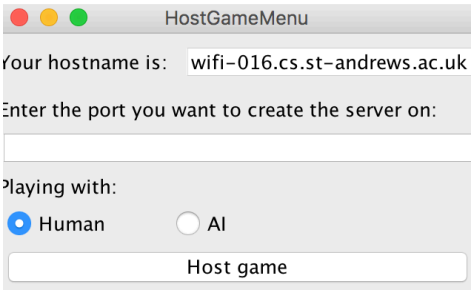
	on that column but a label should be added signifying how many additional pieces are on that column, the label should update when more pieces are added or removed from that column	column but a label was added signifying how many additional pieces are on that column, the label updated when more pieces are added or removed from that column	

Sample OutPut

Main Menu

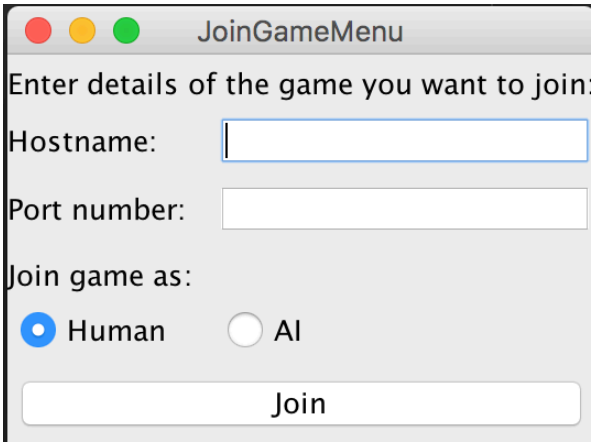


Host Game

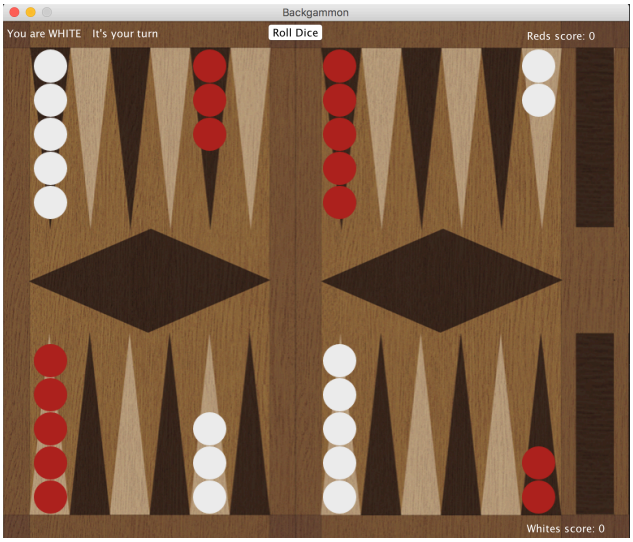


Join Game

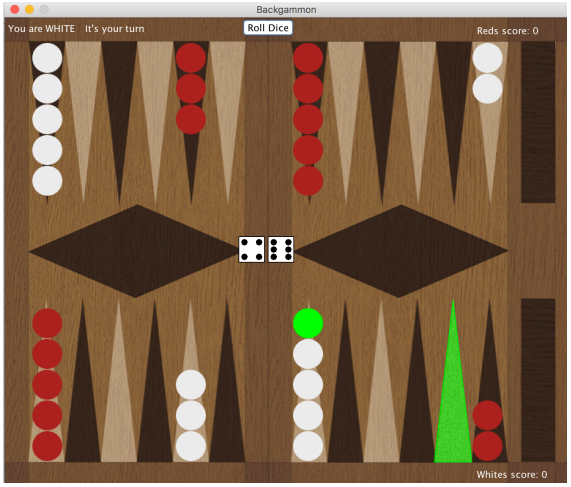
1 April 2016 CS1006 Practical 2 Student ID: 150012773



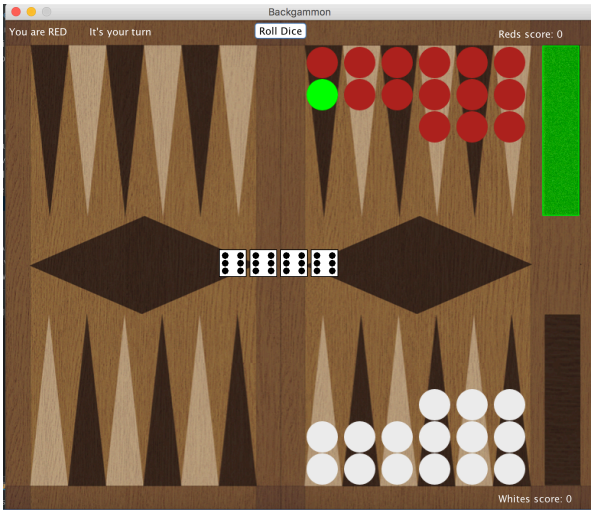
Board

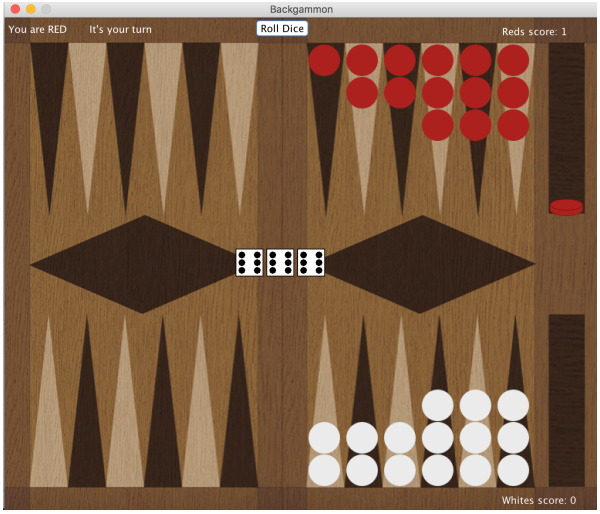


Highlighted counter + Highlighted Available Moves



Born Off Piece





Evaluation

The program we designed meets all the initial requirements successfully as well as incorporating various different extensions. After extensively testing the core functionality of the program, we are pleased to report that the program works as expected and is capable of handling erroneous conditions with ease.

The results from the AI were satisfying and when tested against other AI's out AI commonly won. The speed of the AI was still acceptable when looking at 3 moves ahead.

Conclusion

To conclude we am happy with the solution we have created to the given problem. The code is efficient and neat, providing all the required functionality and more.
If we had longer we would have liked to implement using multiple threads for the AI to improve the speed of the AI. We also considered making a neural network for the AI which would allow it learn.

Individual Contribution

For most of the code we worked together. We used peer programming for the creation core functionality. I mostly worked on the UI and AI. My partner mostly worked on the game logic and networking.