**Artificial Intelligence Final Project Report - Battleship**

**The Game:**

Battleship is a game, generally played by two players, where each player has five ships: aircraft carrier, battleship, submarine, cruiser and destroyer. Each player chooses where to place their ships on the game board. Each ship takes up a certain number of spaces (coordinates) on the game board. The game board is a ten by ten grid with rows A - J and columns 1 – 10. Each player has two boards. One of the boards shows their ships. On this board, the player records the moves made by the opponent. The other board starts as a blank representation of the opponent’s ships. The players take turns listing coordinates (i.e. A1) to target on the opponent’s game board. If the move is a hit, the opponent will say so and the player records the move on his representation of the opponent’s board with the appropriate mark, generally a red peg for a hit and a white peg for a miss. If the player misses, the opponent also announces this and it becomes the opponent’s turn. If all the coordinates of a ship are found by a player, the ship is “sunk.” Once all the ships have been “sunk” on either the player’s or the opponent’s board, the game is over and the person with ships left un-sunk is declared the winner.

**Play Styles:**

There are a few different play styles that a player can use in battleship. The first and most simple is random guessing. As the name suggest, random guessing is a style in which each move made by the player is completely random. This style has no intelligent and is not very effective, however, each style of playing battleship, incorporates random guessing at some point.

The most common play style is hunt and target. In this style, the player starts off performing random moves. Once the player finds a ship and the opponent declares a hit, the player knows that above, below, left or right the rest of the ship must lie. The player then targets these areas for the next few attacks. When a second hit is found, the player should continue down the line until they miss a shot. Most players then switch to the opposite side to be sure the entire ship has been destroyed.

The next play style is known as Parity. This is similar to hunt and target, with one exception. Instead of starting with random moves, the player moves on every other coordinate (A1, A3, B2, B4, etc…). This strategy works because the smallest ship is at least two spaces, thus, if every other spot is hit, you’re guaranteed to hit every ship at least once. Once a ship is hit, the strategy moves back to the target mode as described in the hunt and target style.

The final and most advances style of play is the probability density function. This method uses an algorithm to determine the best move to make next, based on the possible locations of remaining opponent ships. The first move is random. Then, at each turn, the algorithm determines all the possible locations where a ship could fit and builds a density graph which details the likelihood of the presence of a ship at any location.

**Our Project:**

The project we chose to do is build an intelligent version of the classic game Battleship. Our goal was to create a game that is nearly impossible to beat (without using the same algorithm). We set up our game as a standard game of battleship, player verses computer. The game is set up just as described above (two boards, five ships, etc…). We implemented a GUI menu for the user to select certain options before playing. Once the options are selected, the menu closes and the game begins based on the selected options. As stated, our goal was to create a difficult game, so the obvious choice for play style was the probability density function, which far outplays the other play styles, however, we added different levels of difficulty. Obviously, the probability density function is hard mode. Easy mode uses simple random moves. Normal uses hunting and targeting without the probability density function. The menu also allows the player to select makes the first move, player or computer.

**Design:**

We created our project using C# in Visual Studios. It is built with multiple C# class files tied together to form the game. The classes each handle a different aspect of the game or gameplay. First we have a menu class. This handles the menu display and menu selections that are made by the user. This is related to the user options class which actually handles the creation of the GUI menu. Next we have a demo manager class. This is for the gameplay simulations that is used for statistic calculation which are handled in the statistics class. We have a board class which handles the creation of the boards for the game. This uses separate class files as a collection. Included for the board creation is a board class, a coordinate class, a lower screen class and an upper screen class. The board class uses the later classes to create the boards in our game. We have a ships class which handles the creation of the individual ships that populate our boards in gameplay. Next we have a Computer player class. This class handles the creation of the computer player. This uses the related AI class, which handles the artificial intelligence of the computers moves. We also have a player class. This handles the creation of the player’s board and the movements made by the player. The player class and computer player class are used in a player manager class, which handles the gameplay interactions between the two players. The final class is the program class. This class determines how to run the program based on the users selected menu options.

The program displays in two main parts. The first is the GUI menu which appears when the user first runs the program. This was created in Visual studios using VisualC#. The menu displays the options for the user to select. It contains three containers, each with their own radio buttons or check box, one button and one lone check box. The first container is the First Move container. This holds the radio buttons for Player and Computer. A user selects one of the options. By default, Player is selected. The next container is the Difficulty container. This is where the user selects the difficulty of the gameplay. This container holds three radio buttons: easy, normal and hard. By default, hard is selected. The last container is for a Debug mode check box. If this is selected, when the program runs, it will output special information for debug purposes. The lone check box on the menu is the Demo mode. If selected, instead of playing the game, the program will prompt the user to enter an amount of games to run. The program will then simulate the games and output the statistics shown in the simulation.

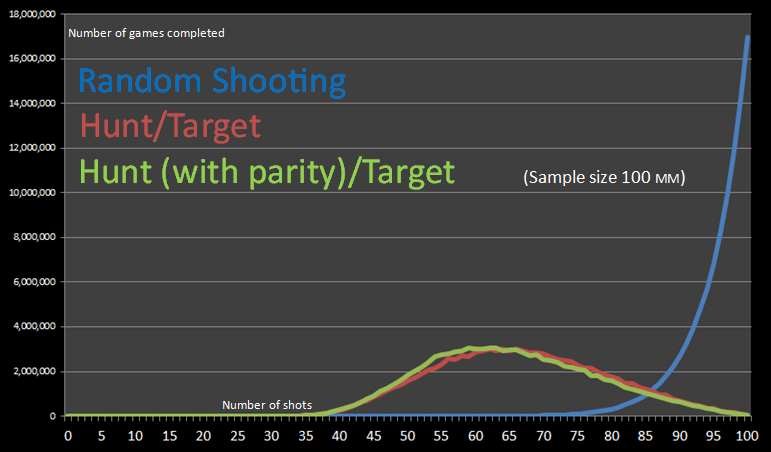
The second display is the actual game display. This is where the game is played and the user makes their inputs. The program displays an updated board after each turn and prompts the user to enter their next move. It also shows information regarding the computers previous move. This repeats until either the player or the computer wins the game.

**Results:**

We set up a test with the program to run multiple games back to back and output data determined from the games. The results showed the amount of moves the computer had to make before destroying all of the ships on the board. We ran six different tests and detailed the result. The first test was on 100 games. This test had a mean move number of 44. The median move number was also 44. The smallest amount of moves the computer made in a game was 23, while the largest was 70. The next test was to simulate 200 games. The results were very similar to the initial 100 game run. The mean and median were both 44, just as shown in the 100 game run. The smallest amount of moves decreased by one in this simulation, bringing the total down to 22. The largest also decreased by one, making it 69. The next test we ran through the simulation was 500 total games. This time the results changed slightly more than the previous run. The mean for this test was 43 total moves, one less than previously observed. The median was down to 42 moves, a two move drop. The smallest move number stayed the same at 22, whereas the largest dropped again down to 67. Although the changes weren’t significant, it seemed at this point that with the more games played, the results were lower. The next run was 1000 games. These results were the opposite of what we previously predicting, mostly increasing overall. The mean and median move numbers both increased back up to 44. The smallest number of moves in a game was the only statistic to decrease this run, decreasing by one down to 21. The largest amount of moves in a game, like the mean and median, increased back up to 70. The results were very similar to the first run, with most of the values the same, the only exception being the smallest number of moves in a game, which was smaller by two turns. Next, we ran a simulation of 5000 games. At this point we expected our results to be pretty consistent, as they have been with previous runs. The mean was, again, 44 moves. The median was down to 43 moves. The most interesting statistic we received in this run was the smallest number of moves needed for the computer to destroy all of the enemy ships. It is important to note that a perfect game (only hitting the ships from start to finish and never missing a single shot) is 17 moves. In this simulation, we had a game be completed in just 18 moves. That means the computer only missed a single shot in the entire game. This is an extremely impressive result. The largest number of moves taken stayed at 70 for this simulation. The final simulation we ran was 10,000 games. As we expected, the results didn’t vary too much in this run. The mean and median were both 43. The smallest move number was again an impressive 18 and the largest number was 71.

After the initial tests, we ran each simulation again to graph the results. Overall, the results from the simulations were quite consistent. The average number of moves it took the computer to destroy all the ships stayed around 44. When comparing this to statistics found online about the number of moves it takes for a game to complete using the other styles, these results are very impressive. With the smallest number of moves being as low as 18, this gives promise that with this algorithm, a perfect game could possibly be achieved. The graphs created from our data are listed below.

Comparing these to graphs found online detailing the shots it takes for the other methods to finish, it shows the probability density function is far superior and on average takes less shots to complete. The graph we used for comparison is below.



(This graph is not based on our own data and was found at http://www.datagenetics.com/blog/december32011/index.html)

As you can see in the graph, in millions of games simulated, little to no games were completed in under 35 moves. With the probability density function, the average number of moves it took for the computer to complete a game was around 44. The graph above shows only a small percentage of games were completed in a similar time using other methods of play. We believe this proves that the probability density function is the most advanced algorithm we could use for the artificial intelligence in our battleship program.

**Division of labor:**

Cory was responsible for the bulk of the C# coding. Seth was not as experienced with C#, so Cory focused on writing or assisting with writing the code. Seth was responsible for building the GUI elements (menu and board) and associated code. Both members were responsible for researching the probability density algorithm as well as other play styles and battleship basics. Seth handled the bulk of the written report and building the simulation data graphs.

**Conclusion and Future Work:**

There were a few elements we wanted to add to the program that we weren’t able to due to time limitations. The first and most obvious would be a GUI for the actual gameplay. Although we did build a simple GUI, we didn’t have time to connect it to our program. With the GUI we would have a cleaner play style and coordinate selection. We also wanted to implement a way to simulate the other styles of play so we could gather our own results for comparison instead of relying on data from the internet.

The program we build has proven to be a tough competitor in the game. While playing against the AI, it is a rare occurrence to walk away the victor. We believe we have managed to create a challenging battleship AI, successfully completing our main goals set for the project.

**Sources:**

http://www.datagenetics.com/blog/december32011/index.html