**COMP813 Artificial Intelligence, Semester 2, 2023**

**AI Project Option A: Battleship Game**

*Finn Boyle, 18034590*

*Introduction:*

For this project, I have created a game of Battleship that is playable against a computer-controlled opponent. My initial plan for this project was to develop the game and train the AI player using reinforcement learning and Monte Carlo tree searching methods, however due to my initial game state implementation, the final product uses supervised learning, heuristic searching with elements of knowledge representation. This project was built nearly entirely by me alone, with some assistance in creating the AI from Coding Cassowary’s Battleship game tutorial series on YouTube **[1]**. I chose to build the project in Python, the game is text based and is played entirely using keyboard inputs.

*Problem Definition:*

To quickly define Battleship, it is a strategy game played without perfect knowledge of the game state. Players have two boards each representing friendly and enemy seas, hidden from the opponent, the friendly ‘sea’ board is for placement of their own ships, and the enemy ‘sea’ board is for recording shots taken against the opponent. Players place their five ships of differing lengths – five, four, three, three, and two – on their placement 10x10 grid. The aim of the game is to take turns “shooting” at co-ordinates on the opponent’s board. The opposing player will then confirm if the shot is a hit or a miss. If the shot is a hit, the shooter will mark on their enemy ‘sea’ board the location of the hit with a red pin, and the opponent will mark their friendly board with a red pin. A white pin is used instead if the shot is a miss. The game continues with players taking turns “shooting” at their opponent’s board until one player has sunk every one of their enemy’s ships, this player is the victor. This is a slightly modified version of the official Hasbro Battleship game rules **[2]**. Rule modifications primarily include players not being required to state which ships have been hit and sunk, to increase difficulty for both the player and the AI, forcing the AI to search the board more rigorously. Another modification of the rules is the need for two boards each. Instead of a ‘ship’ board and an enemy ‘sea’ board, I had the player board reflect their ship placements and the AI shot recordings, and I had the player shot recording board translate new information directly to the AI’s hidden ship placement board. This reduced the total number of shown boards to two total, down from two per player.

The main problem with teaching an AI to play Battleship is making it learn where to place shots if it has not made any hits or has exhausted spaces surrounding a hit. The AI can of course search the board a random pattern, but this makes the game far easier for the player, as the AI will take many more shots than needed to find an enemy ship, often requiring the 100 shot maximum to find all enemy ships if firing completely randomly, as evidenced in a study by DataGenetics (2011) **[3]**. To make improvements, the AI can search the board in a checkerboard pattern, searching every other tile for a ship due to the smallest ship being of length two. DataGenetics **[3]** found that this method reduced the average number of shots down to approximately 60.

While nearby hit searching, random shooting and checkerboard searching were readily implementable, I wanted to expand on these methods with an AI that could determine optimal shot placement based off data collected from previously played games – i.e., the AI would be able to play against the human psyche, learning where humans are most likely to place their ships on a Battleship game board to increase their chances of winning the game. I needed a way to let the AI learn where humans will most likely place a ship on their game board, such that it could beat a human based on human unconscious thought processes.

*Motivation:*

I initially chose Battleship for my AI project because it was something I had not tried before yet felt that it was within my capabilities as a programmer. However, as I began researching the project, I began to feel that previous Battleship AI works did not touch on the specific problem I wanted to solve – making the AI learn from common human ship placements – many projects focused on single game learning, where the AI would make the most optimal shot based on where the previous shot landed rather than learning over time.

My final motivation, therefore, was to create an AI that learn from real historical data of human ship placements every game, every individual game improving its chances of hitting a ship on the player board sooner.

*Methodology:*

To build my Battleship game – with the always-improving AI opponent – I followed the approximate structure of discovering requirements, developing and testing the game and its corresponding logic, developing and testing a basic AI, and finally implementing a more intelligent AI to increase difficulty of the game for the player.

1. *Requirements discovery.*

Before I started on anything, I tried to get an initial idea of what I would need to complete the project. I first needed a game for the AI to be placed into, therefore I started brainstorming required Python functions. These initially included two game boards per player – one for showing to the player and one for processing – and functions for printing the public versions of those game boards, as well as dictionaries of ships for each player which included name, symbol, length, hits required to sink and a sunk status. Other functions included placing ships, base shooting methods, checking for hits, checking for sinks, and checking for wins. Sanitisation of player inputs was also required.

1. *Game board and logic development.*

The previously mentioned features were all successfully implemented, with appropriate but relatively primitive formatting for text outputs. I had thought I may need a third set of boards per player for processing shots to opponent boards, but these were not needed in the final version.

The game would start by allowing the player to place their ships in any order, in any location either horizontally or vertically on the board. Next, the AI ships would be placed randomly across its board. After this, participants would trade shots to each other’s boards until the game was complete, where a winner would be determined, and the code would finish.

1. *Testing the game board/logic.*

Of course, the game building did not happen perfectly from the start, and much testing of the game logic was needed before any AI could be implemented.

The first challenge was to allow any ships to be placed, player or AI. Problems included removing the ability to place one ship on top of another, the ability to place ships outside of the game bounds, and general sanitisation of ship placement input information. Each of these problems were solved before further testing was allowed. All testing was done on the player board, and AI placement methods were ignored for this stage.

Next, testing on the shooting functions was done. To do this, I aimed the shoot function at the player board and fired upon my own ships, skipping the AI code to allow me to focus on the shooting function. Shots needed to change empty sea co-ordinates from “~” to “M” to indicate a missed shot and needed to change any occupied sea co-ordinates from the ship symbol to “H”. Also, the problem of shots on one specific location being repeatable needed to be removed. Board updating was successfully implemented, and repeated co-ordinate shots were solved by requiring the selected location to not display either “M” or “H”, otherwise the shot would need to be retargeted.

After satisfactory testing, board targets were reset to player-to-AI and AI-to-player.

1. *Basic AI development.*

Thanks to the Coding Cassowary on YouTube **[1]**, I was able to implement a basic AI into the game. This AI would shoot across the board in a checkerboard pattern, checking every other co-ordinate for ships – due to the smallest ship being of length two – otherwise the AI would shoot in random locations as a failsafe, if all checkerboard locations had been exhausted. If the AI registered a hit, it would begin searching the directly surrounding seas for the rest of the ship. If a shot registered a hit in a location directly next to the initial hit, the AI would then begin searching for further hits in a straight line extending out from the first two hits until no further hits were detected.

I extended upon this base AI by improving the checkerboard pattern. If the aircraft carrier ship (with a length of five) is not sunk, I made the AI search every fifth square on the board, as logically this ship would be discoverable in this pattern due to its length, and the amount of space it takes up on the board. If the aircraft carrier is sunk, the next longest ship – the battleship, with a length of four – is selected and the AI checks every fourth square on the board. This process is repeated as the larger ships are sunk, until finally only the destroyer is left. Note that both the submarine and cruiser need to be sunk before moving down to checking every other square, as they are both of length three. I retained the function for shooting randomly if no checkerboard spots remained, although it would be seldom used by the AI.

The Coding Cassowary’s AI was built for a program that displayed the game in a GUI (Graphical User Interface), which I adapted to instead work in my text-based game format.

For this section, I also implemented a method for the AI to place its ships in a random pattern across its board. This method was satisfactory from first testing, as it adhered to the same rules as the player for placing ships, meaning it could only place ships in unique locations without overlap, and within the game bounds.

1. *Testing the basic AI against players.*

I tested this AI by simply playing against it, tweaking values as needed. The game was playable, if quite simple. The AI would shoot at the player board in a pattern that was a significant improvement on random shooting, but it wasn’t able to learn from the games it played, and as such suffered – it could not pre-emptively guess where a player might have placed their ships.

1. *Improving the AI with supervised learning methods.*

Now that the game was playable and the base AI was usable as a failsafe, I was able to begin improving the shot selection algorithm with learned strategies. To do this, I created functionality for reading from and writing to an external file that would contain accumulating previous game data. After the player places their ships, their placement data is saved to this external file – a Comma-Separated Values (.csv) file for my implementation. The placement data is saved in the form of a 10x10 grid, with a one representing the presence of a ship, and a zero for a lack of a ship, this information is then added to any information saved in the training data file.

With this new saveable data functionality, I was able to begin improvements on the AI targeting. Before either player takes a shot, the saved data is read in to an extra, private, board from the training data file and operated upon. Every value in the new private board is transformed into a weighted value consisting of the co-ordinate’s integer value divided by the maximum integer value found in the training data grid. With this weighted grid, the AI shot function will now prioritise shots on locations in the game board with a high enough weight in the corresponding weighted board. The shot function would now accept parameters for this weighted board and a required certainty value – which I have set to 85% - to allow for more advanced targeting. The AI will add any locations with a high enough certainty to its potential shot list and make shots onto the board from that list. After firing at a location in the potential target list, that weighted location is set to zero, to prevent the list from filling up with targets already fired at. Should there be no locations above the certainty mark, the AI will relax the certainty requirement a small amount, to allow attempts to continue to be made.

If this function fails, and no shots are detected, the AI may revert to the basic AI, which if that fails in turn, a random shot is made instead.

1. *Training and testing the improved AI.*

Finally, the improved AI was tested many times, allowing the training data file to build up each time.

Since I save ship placement data from the current game before the AI can take a shot, the first test game meant the AI as 100% guaranteed to hit on its first shot. However, this advantage no longer exists, the code cannot automatically reset the training data to all zeroes barring ones for current ship placements.

To increase the rate at which the AI learned, I temporarily stripped functionality from the game. I repeatedly input ship locations into the game, upon which the game would terminate, allowing me to go back and place ship locations in again, saving those new locations once again. Each time, I tried to place ships in locations I thought would be tactically secure, and likely to get me a win against the AI.

After increasing the AI’s knowledge dataset, I returned the game to full functionality, and played normal games against it to gauge its intelligence. While the AI will technically never stop learning, it’s probability database will stabilise after enough games have been played – the value of adding one to cells where a ship is placed will have continually decreasing effects on the weighted board as more and more games are played.

*Evaluation:*

After creating and testing the Battleship game and AI decision making process, I believe my AI can be classified as something that learns over time and is trainable. The player can play games against the AI successfully – the player is able to choose locations to fire at, and the AI is able to determine locations to fire at based on accumulated game data collected over many games. Furthermore, as the player and AI fire at each other, they can sink each other’s ships, and the first one to sink all the other’s ships is crowned the winner. Essentially, the game is functional, and a computer acts as the player’s opponent. This computer can learn from each played game how a human player thinks when placing their ships.

While the game is entirely text based – and therefore visual options are limited – I believe it is a pleasant enough game to play. The AI is acceptable as an opponent, although it is susceptible to bias – especially when trained to play against one player specifically – when new players try the game. The AI does have some quirks, such as searching every co-ordinate neighbour of a hit when to a human there wouldn’t be any likely hits surrounding the area. I chose to leave this in, however, as I was tricked by the AI’s ship placements in some of my test games – I had thought that I had sunk a ship, when it was really two ships placed directly next to each other.

In terms of the type of AI, my Battleship game incorporates supervised learning, heuristic searching, and limited knowledge representation in the form of the board of most likely ship placements.

Supervised learning, as defined by IBM **[4]**, “is defined by its use of labelled datasets to train algorithms that classify data or predict outcomes accurately”, which my algorithm makes use of. Every cell in the weighted grid is a likelihood of the corresponding game grid location containing a part of a ship. The training data file is a collection of real game ship placement datapoints and is used to help make decisions on where to place the next shot. The algorithm therefore incorporates heuristic searching to help it determine where to place any given shot. Heuristic searching being defined by Randall (2020) as “searching the solution space while assessing where in the space the solution is most likely to be and focusing the search on that area” **[5]**.

With the weighted probability board, my algorithm uses a simplified form of knowledge representation – defined as storing data in a database and learning from it over time such that it may become more intelligent over time **[6]**. The weighted board is a historical, numerical record of all previously played games. The AI is able to use this record to store new information and draw on old information to make decisions on currently occurring games.

*Discussion – Limitations:*

This project was certainly not without limitations, major ones being the order of implementation of features, and difficulty finding premade datasets of most common ship placements in Battleship games.

If I were to do this project again, I would change the order of implementation of features. My problem with this is that I created the game logic before creating the AI opponent. Creating the game logic first meant that I was boxed in on how I could implement the AI. Rather than building the game around the AI, I felt that I was building the AI around the game. This limited what kind of AI I could implement into the game. Initially I had planned for the AI to use Monte Carlo Tree searching and reinforcement learning, but my game functions were not prepared to handle this type of searching and no functions were equipped to give rewards/punishments to the AI for good or bad actions. As such, heuristic searching, and supervised learning were the next best options that would fit into my game logic without requiring entire redesigns of the system.

Secondly, the datasets I was looking for do not appear to exist, at least not in the form I need. For my system to work, I need a grid of most common human ship placements that maximise – in the human’s perspective – the chance of winning. In the end, I could not find any dataset such as this, and I could not resort to automatic generation of the dataset as players needed to manually put in ship placements so I could build up a dataset of the human experience in placing ships optimally. Instead, I trained the AI manually, having the game played many times manually – full games and altered games where only the ships could be placed – to build the dataset on my own.

*Discussion – Future Work(s):*

If further work were to be done on this project, my primary suggestions/ideas would be to implement a system for the AI to learn where it could best place its own ships, a method to improve the AI such that it ‘gives up’ on an area of the board after finding no extra hits near a previous recorded hit – as mentioned earlier in this report, and finally to allow the AI to switch strategy mid-game.

Currently, my AI has its ships placed for it, randomly. This project could be improved by making the game more difficult for the player, as the AI could learn where a player is most likely to take shots against it and avoid placing ships in hotspots of common player shots.

Furthermore, the current iteration of my AI will shoot surrounding a ship after getting multiple hits on it. For an example, if the aircraft carrier is hit, the AI will hit the first part, then work its way through the rest of the ship, but once the entirety of the carrier is hit, the AI will continue in that area. It will continue to take shots directly on either side of the ship, and at its tips, chancing that the aircraft carrier is perhaps the battleship and one hit of another ship. I would improve upon this issue by allowing the AI to give up on an area for a time if there have been many unsuccessful attempts at this specific area already. If no further successful hits occur elsewhere for a number of shots, the AI may return to this area.

Finally, I would allow the AI to alter its strategy mid-game. Currently, if the player knows or has access to the information in the dataset, the player would be able to outsmart the AI, placing ships in areas statistically least likely to contain a ship. I could allow the AI to invert the weighted board for a small number of shots if all previous shots result in no hit for the AI, letting it assume that the player might have access to its knowledge base.

*References:*

1. Coding Cassowary. (2022, April). *Battleship AI*. YouTube. <https://www.youtube.com/playlist?list=PLBLV84VG7Md8SgHlXuQXPMJLDvCaWVVQv>
2. Hasbro. (1990). *Battleship*. <https://www.hasbro.com/common/instruct/battleship.pdf>
3. DataGenetics. (2011, December 3). *Battleship*. Retrieved October 6, 2023, from <http://www.datagenetics.com/blog/december32011/>
4. *What is Supervised Learning?  | IBM*. (n.d.). <https://www.ibm.com/topics/supervised-learning>
5. Randall, M. (2020, April 1). *Heuristic Search Part 1: Introduction and Basic search Methods – Matt Randall*. <https://www.lancaster.ac.uk/stor-i-student-sites/matthew-randall/2020/04/01/heuristic-search-part-1-introduction-and-basic-search-methods/>
6. *Knowledge Representation in Artificial Intelligence - JavatPoint*. (n.d.). www.javatpoint.com. <https://www.javatpoint.com/knowledge-representation-in-ai>