Battleship AI

## Hours

Most of my hours included coding both the game logic and the ai used to play the game. At the beginning, I spent a lot of hours planning what my project would be, and planning game logic. I started implementing a battleship program, but after a while of coding it I scrapped my code because I realized it would be much quicker and efficient to alter an already existing game that I could find on GitHub. I ended up forking a project on GitHub *wesleynw/python-battleship,* which was a basic skeleton of the game in python. After forking it, I was able to learn the code quickly, and got to work on implementing an AI for the game rules given. I had to make quite a few modifications to the original game code in order for my AI to work (and to fix quite a number of bugs with the original code). I spent the majority of my time on this project working on the AI and tweaking it so that it could win the majority of the time.

***Overall:***

Hours coding initial project: 4

Hours planning my project out (game logic/AI implementation): 2

Hours fixing forked game/learning it: 7

Hours implementing AI for battleship/debugging: 10

Hours testing and running games: 2

Total: **25 Hours**

## Project Information

My implementation for battleship is simple, but really made me think about different approaches that I could take. I chose battleship for my project because unlike some of the other labs we were given, there is an unknown factor to it. This made me think about the different ways I could try and implement the different AI techniques that we learned about in class in order to win. This aspect of the game prevented me from using things like Monty-Carlo Tree Search or other algorithms because I couldn’t implement them without giving the AI an overwhelming advantage.

#### Goal

My goal for this project was to create an AI that would win most of the time by using a local search type technique. It would make the best possible move given the current game board.

#### Issues

The actual implementation would be tricky with a game like Battleship. As mentioned, I decided that a local search would be the best option, as it seemed to me to be the only true way to simulate a player for battleship. Given that the AI was not allowed to know about the other player’s board, it made it difficult to find a planning strategy that the AI could implement. I couldn’t plan anything without knowing some sort of information.

### Implementation

My implementation drew inspiration from the Bayes Filter lab. I decided the best way to guess where a player’s ship would be is to simply find the square on each move that has the greatest probability of having a ship. At the beginning of the game, I make a list of the other player’s ships, and for each length I find all the possible squares it could be in. For each window of N length (N being the length of the ship I am searching), I add 1 to each square. If a window contains a miss, I know that the ship I am looking for would not be there, so I don’t add anything. If the window contains a hit, then its very probable that the ship is in that window. Multiple hits in the same window increase the value exponentially. This ensures that each move has the greatest probability of hitting a ship. This way also ensures that at each move, the AI also narrows down the search for the other player’s ships.

##### Interesting Behavior

This implementation of my battleship AI did leave to some interesting behavior and taught me some things about the battleship game. I noticed that at the very beginning, the AI would make the same first few moves. This is because there is a greater probability that there is a ship in the middle of the board than there is on the edges. A lot of the time it would find an enemy ship within the first 3 moves, but if not, the AI quickly set itself up to narrow down each ship one at a time starting with the biggest ship. While my game did not quite follow the rules of battleship that I enjoy (if you get a hit, then you get to go again), it set itself up for the most likely way to win, that is targeting the biggest ships first and then going down to the smaller ships (See GamePidgeon version of Sea Battle for more details). Even without these special rules that give the best strategy for a different type of battleship game, it still would win most of the time.

### Results

I ran 50 simulations, where both my wife and I played the AI in a game of battleship. It took a long time to gather this data, and one thing I regret is not having an option that would simulate the game automatically. It would make it a easier to gather data and see where it could improve. Overall, it beat me 24/25 times and my wife 20/25 times (she is better than me at the game). I found that even when I would try and exploit the AI by putting my ships to the side and on the corners, it still ended up beating me. Its strategy is very methodic and does seem to win the majority of the time, which makes me think that battleship, while luck is involved, is a very mathematically optimized game.

## What I Learned

I would say that the biggest thing I learned was that we have a lot of different techniques for AI that we learned in the course, and that you can use samples of all of them to create AI depending on your circumstance. Not only did I realize that not all of the different algorithms would work in this situation, but that really mixing and matching the different parts of the techniques we learned in this class was truly the way that we can make AI for a lot of different scenarios. I am super excited to continue my learning in AI techniques, and would love to continue to study these algorithms in more detail.

One thing I wish I could do in the future is expand this project to play the Sea Battle game on iOS, so that I could play with my friends and have the AI automate my entire game. That way I can turn his simple game AI into something that is more easily used. This will be where my research will take me in the future.