Aniruth Narayanan

Thought Process: Battleship

Introduction

This project was inspired by the game Battleship.

The best part about computer science, for me, is algorithms and the thought process to design them – whether it is symbolic artificial intelligence as in rule-based procedurals or data-driven artificial intelligence as in machine learning[1]. Symbolic AI is used to code many chess engines, such as Stockfish, while data-driven AI is used to code other chess engines (regarded as superior) such as AlphaZero. Data-driven AI uses many instances of test data to effectively train the computer to place its own values by using a cost function to adjust the weights and biases[2], while symbolic AI follows the instruction of the user. Humans may place certain value on specific things, but computers can counterintuitively be trained to go against such principles to win the game.

For the most part, the discussion focuses on games and not actual tasks due to the ideas of Moravec's paradox, which state that it is easier to teach a computer things we would consider to be "advanced" – such as playing chess – than things we consider "simple" – such as comprehension.

With these concepts in mind, I wanted to tackle a game using symbolic AI in terms of determining my own algorithm and then coding it into a game.

The game I wanted to tackle was initially chess, before I realized just how complex chess is and that it would require me to write code for a very, very long time. Part of it is the complicated rule set, including moves like en passant, but also that human input is involved. I didn't want to create an algorithm for an overly complex game, but I also didn't want to code a game that required no strategy like rock-paper-scissors. Thus, I settled on Battleship – but only the computer guessing, not the player guessing.

I consider this project to be a bit confusing (as evidenced by the extraordinary time it took me to debug), so I've split this into a few sections. Most of it won't make sense without the actual files pulled up. First, I'll talk about the board class, then the ship classes, initializing the game from user input, the strategic algorithm explained conceptually, and finally the code itself.

Building the Board

There were a few things that I knew going into the project from its initial conception. I wanted to try my hand at object-oriented programming, so each ship and the board was going to get its own class that I was then going to use and manipulate in the main class. The board would maintain all the hits and misses as well as the locations of each ship, and then the main class would try the algorithm. I had a couple of ideas about the algorithm itself, but I'll talk about those in the algorithm section itself.

[1] These terms are descriptions are used in the book "DRIVERLESS Intelligence Cars and the Road Ahead" by Hod Lipson and Melba Kurman on page 76, chapter 4. There are no doubt countless more subdivisions and categories for something as complex as AI, but for the almost inconsequential discussion here, this more than suffices. It's an excellent book that I highly recommend reading, by the way.

[2] This is a source linking to 3Blue1Brown's video series on neural networks – an excellent introduction to the vast field of machine learning.

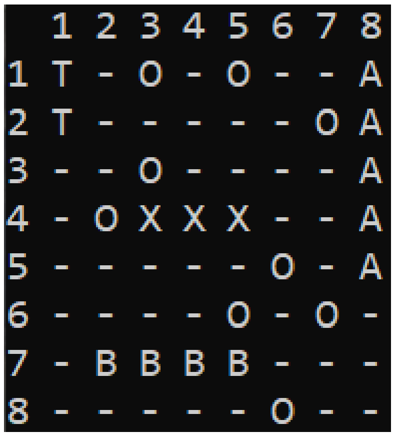
The Board object consists of three methods. The first was the only one that I had planned – an initializer that has a ".display" attribute that is a two dimensional array. My only prior experience coding (in addition to the triangulation algorithm I coded) at this point is some brief experience with Java, so much of this was quite difficult especially in terms of syntax. This is created with "-" throughout to signify no action or ship in that location in the array.

The next method was the printBoard method, which was quite simple – it printed the board. Printing the array directly did not yield the format I wanted, so I decided to go ahead and create a dedicated method that is probably too long. It begins by printing the labels on the very top of the board for the user to see, then printing out a label for each row and then printing the actual values within each marker within the array.

There are two features of significance that I'll elaborate on (the rest should be self-explanatory as I made sure to include a lot of comments in each file): first, there are counters being used, not the actual r and c themselves. This is because I kept coming into errors since the r and c took on the values of an array and the "-" or the other placeholder, which could not be used to access specific coordinates within the game board's two dimensional array. The counters allowed me to get around this and provide labels as well. Second, I used end statements within many of the print statements. This is because the print statements would create new lines by default, so this allowed me to get around that by spacing them out until an entire row was printed.

The last method related to the win condition for the entire game. In order for the computer to have achieved its task, it must have removed every ship – which is what the checkboard method does. It cycles through every value in the game board's array to determine if there is any ship (each is represented by a letter). This returned 0 if the win condition was not met and a 0 if the win condition was met.

Overall, the board method took a few hours simply due to a lack of understanding with arrays, but once I did some research, I was able to put it together. I have attached a picture of what it looks like below:



```
  1 2 3 4 5 6 7 8
1 T - O - O - - A
2 T - - - - - O A
3 - - O - - - - A
4 - O X X X - - A
5 - - - - - O - A
6 - - - - O - O -
7 - B B B B - - -
8 - - - - - O - -
```

Figure 2-1

The O is a miss, the X is a hit, and the other letters represent ships.

Floating the Ships

Each ship (the tugboat, submarine, battleship, and aircraft carrier) also got its own class. They each had an initializer that defined their letters, but the main feature of each class was that they updated the board with their locations. I did not plan this initially, but this makes more sense considering the algorithm.

The algorithm has two parts: first, it creates the game environment – asking the user how many rows and columns they would like the board to have and then to specify the locations of every ship. However, imagine entering five coordinates to place the carrier. It would be tiring and quite pointless – since there are 4 ships, it amounts to 14 coordinates being entered in. Instead, the algorithm looks at the starting and ending position of each ship and then generates the coordinates in between. I'll use the battleship as an example below.

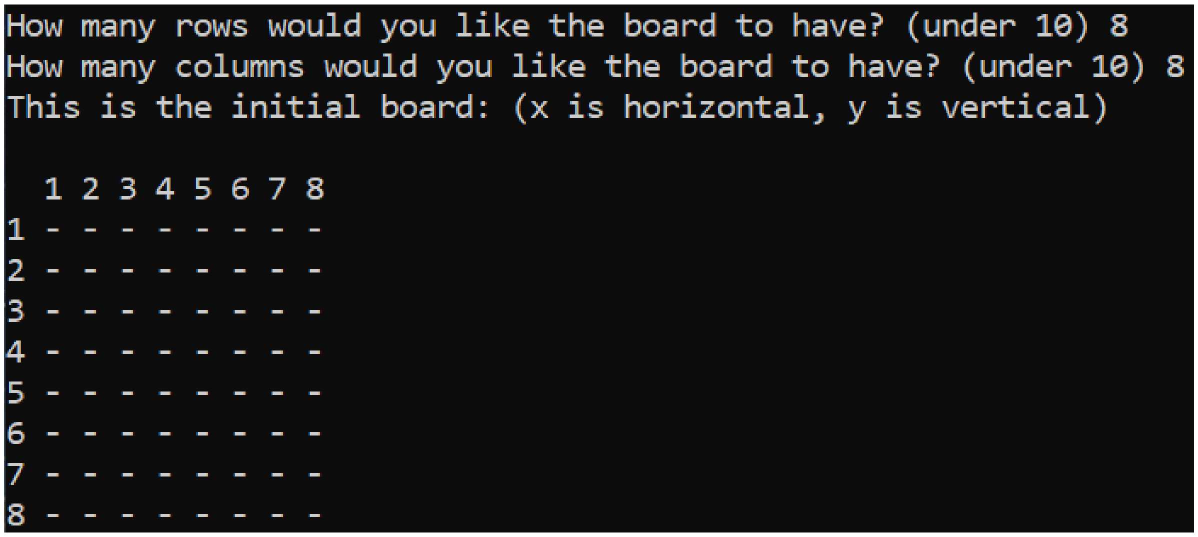
Let's say the user enters a start coordinate of (1,2) and an end coordinate of (5,2) for the aircraft carrier which has a length of 5. One of the requirements is that each ship be placed either horizontally or vertically. In this case, it is horizontal, as the y-coordinates of 2 remain the same. The method updateBoard for each ship checks this. If the y coordinate is the same, it then takes the game board, goes to the locations, incrementing along the axis that is changing, and then ending with the end coordinate. Thus, the board then has each ship's letter, storing its location. The exception to this is the tugboat, as it has a length of only 2 and its start and end coordinates that the user specifies are the only coordinates.

This part of the program was not too difficult either.

Initializing the Game Environment

The main program itself has two parts. The first part will be discussed in this section where the game environment is created – excluding the functions at the top of the program that make more sense in the context of the algorithm.

The program begins as normal – importing the necessary classes. After creating the ships and taking in the user input for the rows and columns, the game board is created with the input and then printed for the user to see initially, shown below in Figure 4-1:

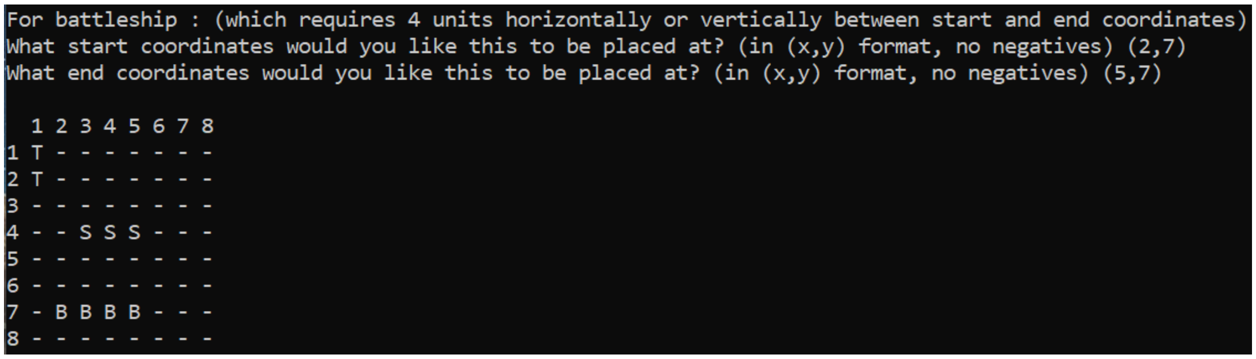


```
How many rows would you like the board to have? (under 10) 8
How many columns would you like the board to have? (under 10) 8
This is the initial board: (x is horizontal, y is vertical)

  1 2 3 4 5 6 7 8
1 - - - - - - - -
2 - - - - - - - -
3 - - - - - - - -
4 - - - - - - - -
5 - - - - - - - -
6 - - - - - - - -
7 - - - - - - - -
8 - - - - - - - -
```

Figure 4-1

The previous approach to placing the ships within the game board is now undertaken. Everything is stored in a two-dimensional array called locations, with the first array representing the x coordinates and the second array representing the y coordinates. A loop is used to re-use the prompts, adjusting each time based on the type of ship and then appending the user's input to the locations array. After each pair of start and end coordinates is used, the updateBoard method is called on each object and then the board is reprinted for the user to see. This repeats for each ship, so that the user can see where they have placed the ships, shown below in Figure 4-2:



```
For battleship : (which requires 4 units horizontally or vertically between start and end coordinates)
What start coordinates would you like this to be placed at? (in (x,y) format, no negatives) (2,7)
What end coordinates would you like this to be placed at? (in (x,y) format, no negatives) (5,7)

  1 2 3 4 5 6 7 8
1 T - - - - - - -
2 T - - - - - - -
3 - - - - - - - -
4 - - S S S - - -
5 - - - - - - - -
6 - - - - - - - -
7 - B B B B - - -
8 - - - - - - - -
```

Figure 4-2

As a quick recap of what has been done up to this point: the board was established with methods to print the board to the screen and to check if there are any ships left, classes for each ship, a board built custom to the user's row and column requirements, and ships placed according to where the user would like. Now, the algorithm for the computer to figure out where these ships are begins.

Writing the Strategic Algorithm: Top-Level

The computer, since it is storing the game board, could do some things to instantly win. For example, it could search for the ships and then replace them with hits, or it could use the letters from each hit to determine the nature of the ship and then determine how many more hits are needed. However, a real user playing the game in trying to guess where each ship is located would be unable to do this, so any advantage a computer would have over human merely because it is a human is considered cheating and cannot be a part of the algorithm.

With this in mind, I will first explain the methodology behind the algorithm in general and then go through the specifics of each method and section within the code itself[3].

How is the game of Battleship played? Well, at first, one doesn't know where to guess on the board. So, a point is picked randomly, hopefully a hit. In the event it is a miss, however, another point is picked randomly, and so on until a hit – and a ship is found. Additionally, once a point is guessed, that point is never guessed again – for that would be pointless, as the outcome of that location is already known.

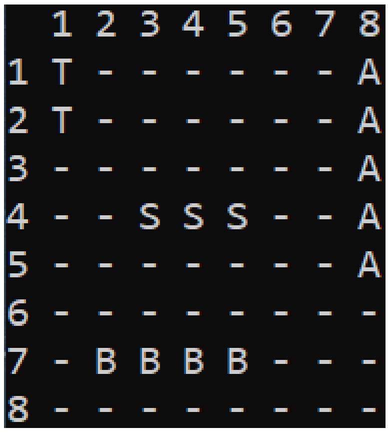
[3] This part will only make sense in reference to the original file.

But when a ship is found, suddenly, the guessing is no longer random. Since every ship must have length of at least 2, there is now a relatively straightforward pattern that can be used. Ships cannot be diagonal; therefore, there is one point that is directly adjacent to the hit that has another hit. That point must be searched for. After checking every direction, making sure that the point is both in-bounds and has not already been guessed[4], it is then checked to determine if it too is a hit or a miss. If it is a miss, the "strategic" guess cycles around, trying to find a hit; if it is a hit, then next process begins.

Up to this point, two hits have been found – the first guess, which I will refer to as "random", and the second, which I will refer to as "strategic". When two hits are found, they form one line upon which the ship must lie. Continuing to check along the direction that was used to get from the random point to the strategic point, a new point, which I will refer to as the "direction" point, is found. If it is a hit, that's excellent! The process continues for a maximum of five, which is the longest ship – the aircraft carrier. However, if it is not a hit, then there is still another scenario that must be accounted for – checking the point that is in the opposite direction from the random point to the strategic point immediately next to the random point. If that is a hit, the checks continue until there is a miss – but there is no need to check the other side, as there was already a miss. In the initial process, assuming the direction point (which is always adjacent to the strategic point) was a hit, and then the next points are a miss, the change in direction to checking next to the random point occurs – but once that switch occurs due to a miss, a counter-switch is not needed again. The entire process requires a maximum of 6 guesses – the random, strategic, direction, and then three more guesses along the line which would be an aircraft carrier sunk after the direction of guessing was switched.

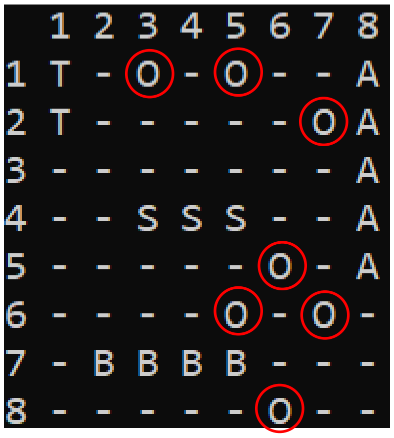
That paragraph was no doubt confusing; it confused me as I was writing it. Unfortunately, such is the nature of algorithms. However, have no fear – I will try and explain this a bit better using some images and then a discussion of the algorithm itself.

The first part of the algorithm is the random guess. The computer, just like a human, does not know where the locations of the ship are. Thus, it must randomly choose a point and then determine if that is a hit or miss until a hit is found.



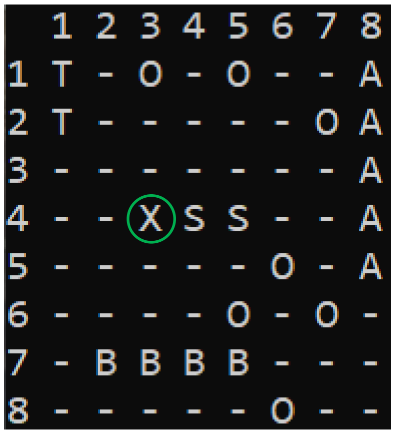
```
  1 2 3 4 5 6 7 8
1 T - - - - - - A
2 T - - - - - - A
3 - - - - - - - A
4 - - S S S - - A
5 - - - - - - - A
6 - - - - - - - -
7 - B B B B - - -
8 - - - - - - - -
```

Figure 5-1



```
  1 2 3 4 5 6 7 8
1 T - O - O - - A
2 T - - - - - O A
3 - - - - - - - A
4 - - S S S - - A
5 - - - - - O - A
6 - - - - O - O -
7 - B B B B - - -
8 - - - - - O - -
```

Figure 5-2



```
  1 2 3 4 5 6 7 8
1 T - O - O - - A
2 T - - - - - O A
3 - - - - - - - A
4 - - X S S - - A
5 - - - - - O - A
6 - - - - O - O -
7 - B B B B - - -
8 - - - - - O - -
```

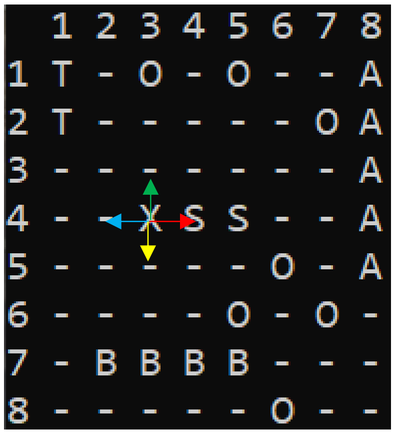
Figure 5-3

[4] This poses no problem. Once a hit is found, the program does not guess randomly again until the entire ship is sunk.

The above three pictures show the random guessing process. Figure 5-1 is how the board looks immediately after the user inputs the start and end locations of each ship; Figure 5-2 represents a few misses at a variety of points (at (3,1), (5,1), (7,2), and so on). Since each is a miss, the computer guesses randomly until a hit is found which is Figure 5-3, displaying a hit on the submarine ship at (3,4), represented by the green circle.

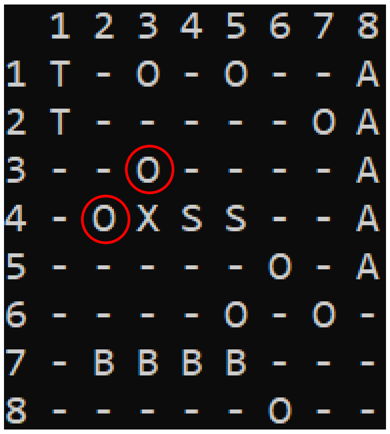
Up to this point, the computer has done a few things. First, it has initialized the board according to the user's wishes; second, it has stored every miss to prevent guessing a miss again, and third, it has stored that there is a hit at (3,4). Now that there is a hit, the computer does *not* generate another random point until the entire ship is sunk.

Every ship has a minimum length of 2; as such, there must be one point in either of 4 directions (left, up, right, or down) where there is another hit on the same ship. Now, the algorithm finds the "strategic" point. The algorithm begins by checking the left direction and determining if it is a hit or a miss. If it is a hit, it proceeds on; if it is a miss, it checks the next direction, which is up. The same procedure applies – if it is a hit, the next steps begin; if it is a miss, it checks the next direction, which is right, and so on for each direction.



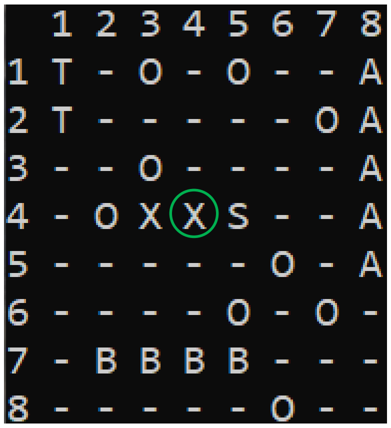
```
  1 2 3 4 5 6 7 8
1 T - O - O - - A
2 T - - - - - O A
3 - - - - - - - A
4 - - X S S - - A
5 - - - - - O - A
6 - - - - O - O -
7 - B B B B - - -
8 - - - - - O - -
```

Figure 5-4



```
  1 2 3 4 5 6 7 8
1 T - O - O - - A
2 T - - - - - O A
3 - - O - - - - A
4 - O X S S - - A
5 - - - - - O - A
6 - - - - O - O -
7 - B B B B - - -
8 - - - - - O - -
```

Figure 5-5



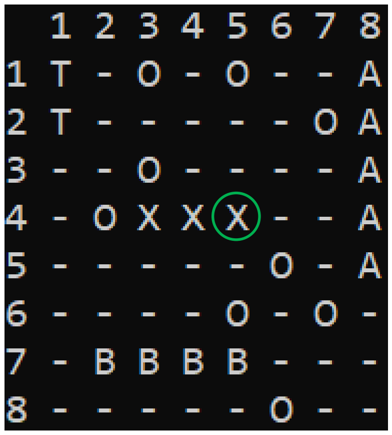
```
  1 2 3 4 5 6 7 8
1 T - O - O - - A
2 T - - - - - O A
3 - - O - - - - A
4 - O X X S - - A
5 - - - - - O - A
6 - - - - O - O -
7 - B B B B - - -
8 - - - - - O - -
```

Figure 5-6

Figure 5-4 is a continuation of Figure 5-3 except it now uses 4 arrows – blue, green, red, and yellow – to signify each of the possibilities that the next hit for the same ship could be. The algorithm always begins by guessing to the left (blue), then up (green), then right (red), and then down (yellow). In this case, as depicted in Figure 5-5, it is a miss at (2,4). It then checks above the random point at (3,3), which is also a miss. The next point to the right at (4,4) is a hit, which is marked with the green circle in Figure 5-6.

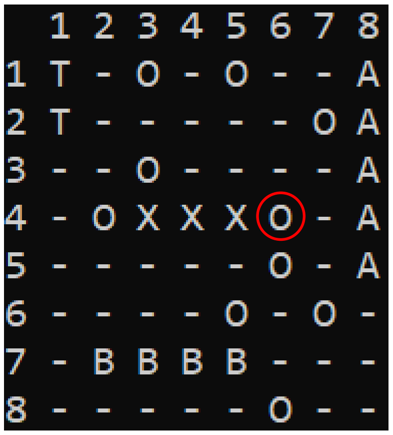
The computer can now skip over a few steps thanks to storing every point that has been guessed. Normally, the computer knows that the ship must line horizontally along the line y = 4, and it continues checking in the right direction until there is a miss. Once there is a miss, it then flips to check the other direction – which is unnecessary in this case as it has already tested (2,4).

The next point found is the "direction" point, since the direction is known. Continuing in the same direction, it then determines if there is a hit or a miss. If it is a hit, then it continues onwards; if there is no hit, it then flips the direction and checks in the opposite direction to the initial random point. However, that point has already been tested (it would be (2,4)), so there is no need to do that in this case once (6,4) is confirmed to be a miss.



```
  1 2 3 4 5 6 7 8
1 T - O - O - - A
2 T - - - - - O A
3 - - O - - - - A
4 - O X X X - - A
5 - - - - - O - A
6 - - - - O - O -
7 - B B B B - - -
8 - - - - - O - -
```

Figure 5-7



```
  1 2 3 4 5 6 7 8
1 T - O - O - - A
2 T - - - - - O A
3 - - O - - - - A
4 - O X X X O - A
5 - - - - - O - A
6 - - - - O - O -
7 - B B B B - - -
8 - - - - - O - -
```

Figure 5-8

The above pictures illustrate the final parts of the algorithm. Figure 5-7 is the “direction” point which determines if the direction needs to be switched or the line can be continued. This continues until there is a miss or the longest ship, the aircraft carrier with a length of 5, has been sunk. Figure 5-8 shows the miss of the “second direction” point, so called because it is the second point after the initial “direction” point. Now that there is a miss on that end, and a miss at (2,4), the ship can be confirmed to have been sunk.

There are a few other cases that the algorithm accounts for that is not shown here. The maximum depth it checks for is four direction points (so a total of 6: random, strategic, direction, second direction, third direction, and fourth direction) to account for hitting an aircraft carrier in the middle, going one direction, getting a miss, and then flipping direction until the whole ship is sunk.

Once the ship is sunk, the next random point is found, repeating the process all over again. However, with every point that has been guessed as either a miss or a hit, the same point is never guessed again. This continues until the win condition has been satisfied which uses the check board method from the Board class which happens periodically throughout the algorithm.

Now that a basic understanding of how the algorithm works in the abstract has been established, a more in-depth exploration of the actual algorithm can proceed.

The Code

This section will focus only on the main class (the algorithm), as the rest of the code has been explained in previous sections. I will assume that if you have made it this far, you have a basic understanding of how the algorithm works. I have included many comments for just about everything, in the file so I will just point out a few things that may be confusing at first glance.

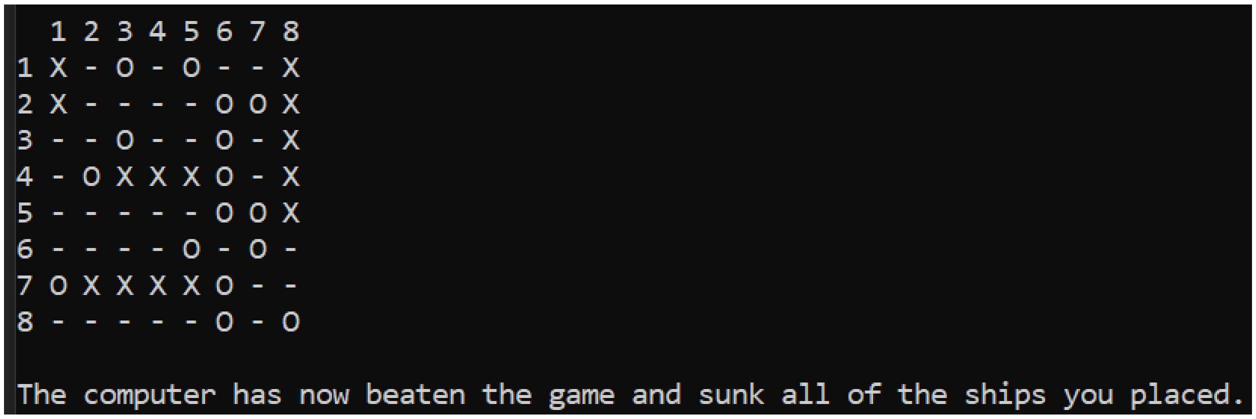
First, there are a few initialized variables for any of the verification and checking to occur. These have a few key definitions. The only one that might be a bit confusing is the alrGuessed array (in line 125). The reason why it has -2 in each array is that one of the methods looks at the length of the alrGuessed array and I didn't want the length method to come back as 0 considering the range function. I doubt that it matters but that is just a habit of mine, I suppose. -1 will be used later, so I used -2.

Both the stratGuess and checkInDirection methods appear to be very similar – because they are very similar. Each looks at different directions, adjusting the input coordinates to match that direction while returning the same direction. However, at the bottom, it does return [-1, -1, -1] which can be a bit confusing.

What happens if a point that is around the hit is outside the realm of the board? That is why the second condition accounts for whether it is in bounds of the specified number of rows and columns. If no points work, it returns a -1, which is then used whenever a direction is flipped – which is why it allows for the point to be either a miss or to be nonexistent if it has a value of -1 since that is what is returned. That is also why each is checked to see if it is positive before continuing onward in the code.

In the event that two misses are found, the code simply breaks to go all the way through the strategic part of the algorithm back up to the generation of a new random point, which is why there are so many "if" statements but quite a few "elif" statements.

As a last note, something to keep in mind is that while the user inputs numbers beginning from 1 up to the number of rows/columns (as in counting numbers), the computer views everything as going from 0 up the number of rows/columns minus 1. This makes some of the random math and indexing a bit easier.



```
  1 2 3 4 5 6 7 8
1 X - O - O - - X
2 X - - - - O O X
3 - - O - - O - X
4 - O X X X O - X
5 - - - - - O O X
6 - - - - O - O -
7 O X X X X O - -
8 - - - - - O - O

The computer has now beaten the game and sunk all of the ships you placed.
```

Figure 6-1

When everything is finally beat, and all the ships have been taken down, then the computer stops the larger while loop as the win condition has been satisfied, prints the board one last time for the user to see as shown in Figure 6-1, and then informs the user that the computer has triumphed over the challenge of beating the game.

Areas for Improvement

As with all programs, there are still ways that this could be further refined.

1. Much of this code is most likely inefficient and redundant, both in terms of design and syntax. For example, if an aircraft carrier has been sunk, it isn't necessary to test for a fifth hit after getting

four hits in a row. However, this just proved to complicate an already overcomplicated and length algorithm.

2. One key area this lacks in is in relation to what the user can specify. In fact, I explicitly state that things like the number of ships could be controlled by the user. However, creating multiple objects based on the number of ships and then accounting for space would be quite difficult. The same reasoning applies to the bounds of user input – instead of using many loops to allow for the user to provide correct input, the program just ends (due to its sheer size at being over 250 lines of code in just the main file).
3. Some of the code is just convoluted. For example, some things are stored as (x,y), while the board looks at (y,x). The game board stores its values starting at 0, while the user inputs values starting at 1.

For now, as usual, I'll leave these to another time.