To Catch a Battleship

New Mexico

Supercomputing Challenge

Final Report

April 11th, 2021

Team 60

Media Arts Collaborative Charter School

Team Members

Tyler Wyskochil

Teacher

Tanya Mueller

Mentors

Creighton Edington

Harry Henderson

Executive Summary

Almost everyone has played the game battleship at some point in their life. When we played however we probably only shot at random, or specific locations that our friends liked to use. For my project the question I wanted to answer was what shot pattern would sink the enemies ships in the least amount of turns in the game Battleship. By simulating a game of Battleship I was able to find which patterns were the best when in different situations including what finds something the fastest or what found the smaller ship more often.

My multiple running hypotheses include my null hypothesis where all of my search patterns have the same effectiveness. My second hypothesis where having a gap of one was most effective. My third hypothesis where having a gap of two was the most effective. My fourth hypothesis was if the gap of three was the most effective. My final hypothesis was if the gap of four was the most effective.

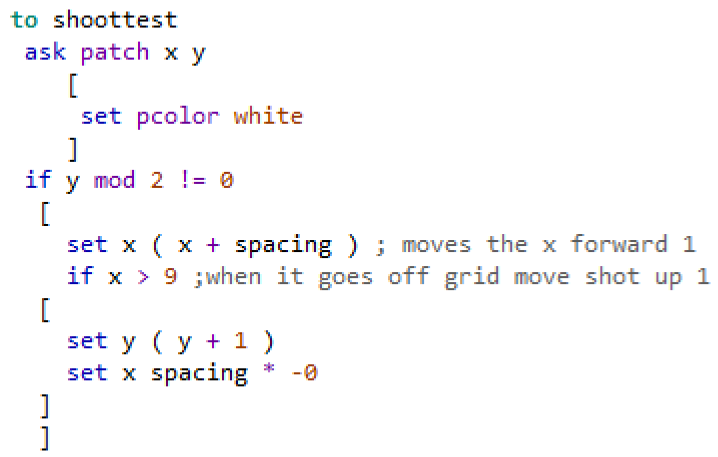
The project was done by randomly spawning four boats. After running each shot type 10,000 times, I was able to see which pattern did the best in different types of situations. The results for what pattern found all the boats the fastest showed that my null hypothesis was correct. There was no statistical difference between the effectiveness of the patterns. However, the results also showed that by searching small you find small, and by searching large you find large. With each search pattern the bigger the gap the higher the chance that you would find the larger ships first. This worked the same with the smaller pattern as the smaller boats were found significantly more often than any other pattern.

Introduction

With this being my first year coding I was having trouble coming up with something to work on for this project. I simply didn’t understand the capabilities and limitations of Netlogo. I decided it would be best then to go through some of the archives older projects done during previous supercomputing years to get an idea of what to do. It was through searching these projects that I found someone from my school who had worked on a project about the best way to find boats in the game Battleship. As someone who always used Battleship as their go to game I was interested in the best way to win. After looking through it however, I found that the project was never completed, so I decided to do the project justice and do it myself. The program simulates a game of battleship by creating a 10X10 board and spawning four boats. I chose to spawn four boats instead of five because having two ships of the same length negatively affected the results. The program then shoots in one of the set patterns. The program then runs the selected pattern until it hits a boat, where it will shoot around the hit location in an attempt to find the boat. Once it destroys the boat it hit, it will continue on with the selected pattern until all boats are found.

Description

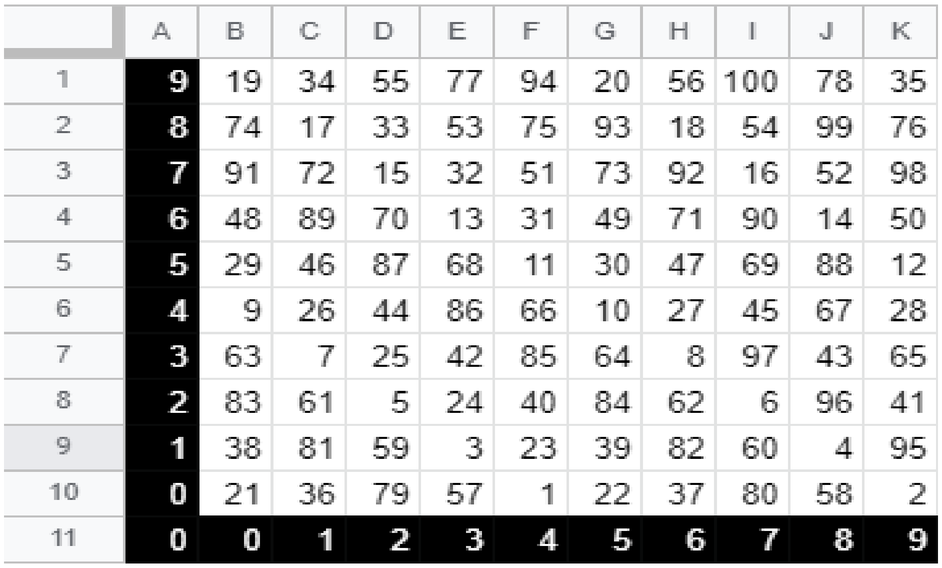
The first hurdle I had to overcome was figuring out how to do the shot patterns. I tried several ways to try to set up the patterns by having a single mathematical procedure to do the shots. For example I tried using mod as a way to separate my shot patterns. Sadly I was never able to get that to work so instead I was forced to come up with another method.



```
to shoottest
 ask patch x y
    [
     set pcolor white
    ]
 if y mod 2 != 0
  [
    set x ( x + spacing ) ; moves the x forward 1
    if x > 9 ;when it goes off grid move shot up 1
  [
    set y ( y + 1 )
    set x spacing * -0
  ]
  ]
```

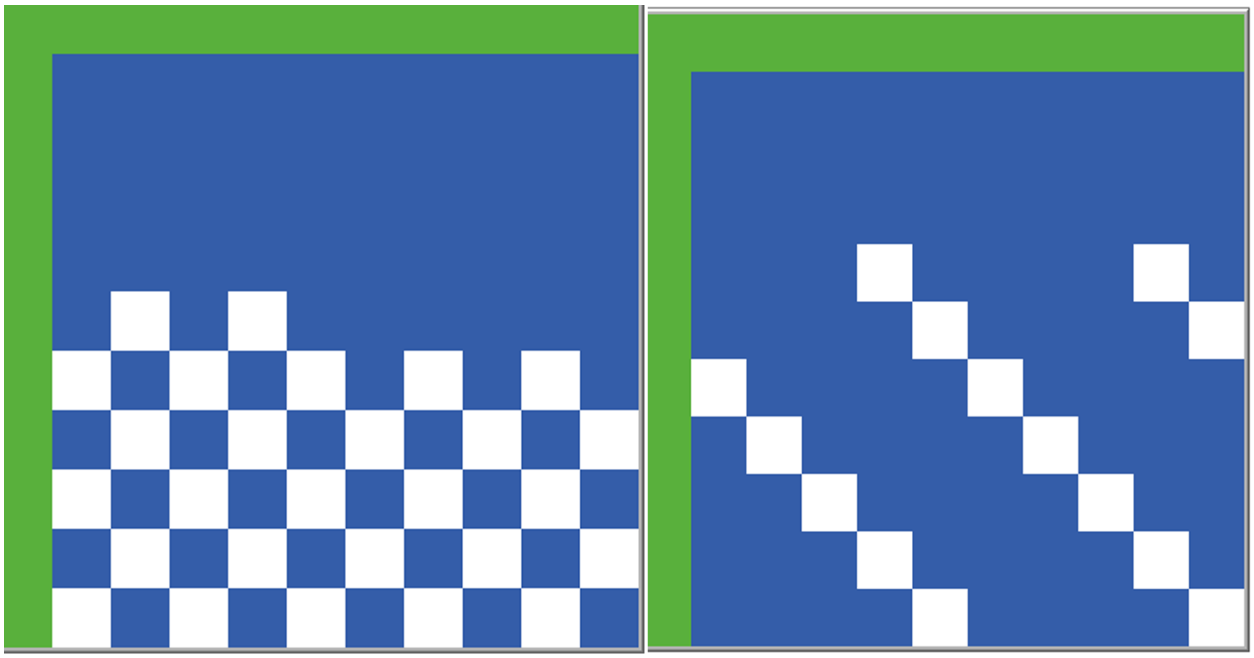
. This new method used text documents that would hold the exact shot positions of the search patterns. A text file was made for each specific pattern. I then made a block of code that would read the text document and assign the numbers to a list that would be used for the pattern. For random I simply just used “select one-of patches” to determine what patch would be targeted.

How I created the pattern for the 4-skip pattern

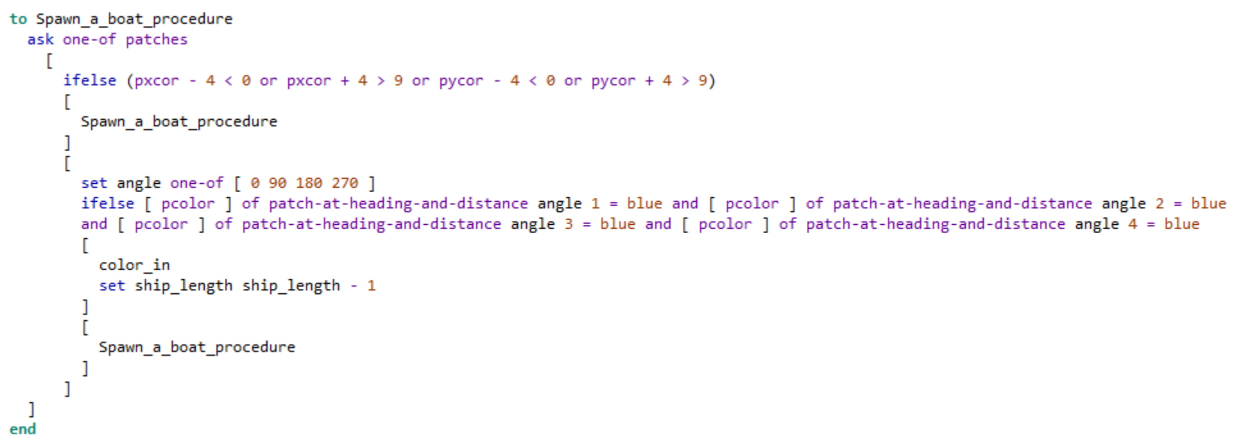


	A	B	C	D	E	F	G	H	I	J	K
1	9	19	34	55	77	94	20	56	100	78	35
2	8	74	17	33	53	75	93	18	54	99	76
3	7	91	72	15	32	51	73	92	16	52	98
4	6	48	89	70	13	31	49	71	90	14	50
5	5	29	46	87	68	11	30	47	69	88	12
6	4	9	26	44	86	66	10	27	45	67	28
7	3	63	7	25	42	85	64	8	97	43	65
8	2	83	61	5	24	40	84	62	6	96	41
9	1	38	81	59	3	23	39	82	60	4	95
10	0	21	36	79	57	1	22	37	80	58	2
11	0	0	1	2	3	4	5	6	7	8	9

Partial 1-skip on the left and partial 4-skip on the right.



The next step in creating my code was to create the boats. I initially created a simple while loop that would randomly choose a patch then create a boat in a random direction then lowered the value that determined the boat size. This original idea had many issues. The first issue was due to a problem with the recursion it would sometimes make too many or too few boats. It also had a major issue with it trying to call “nobody”. I attempted multiple different ways to fix this problem. I tried limiting the spawn area so that a nobody patch could never be called. I also tried setting up an if statement that would reset the program if nobody was called. I attempted to get help through the coding help session but it was never resolved.

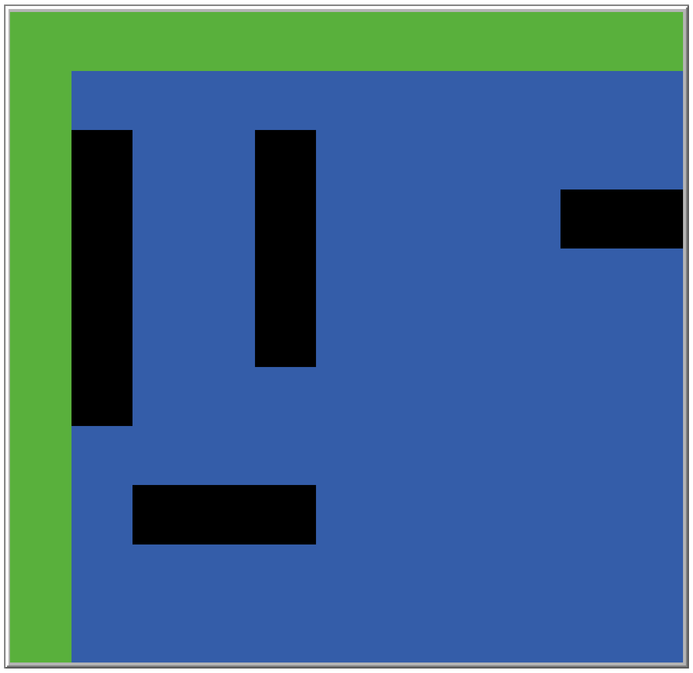


```
to Spawn_a_boat_procedure
  ask one-of patches
    [
      ifelse (pxcor - 4 < 0 or pxcor + 4 > 9 or pycor - 4 < 0 or pycor + 4 > 9)
      [
        Spawn_a_boat_procedure
      ]
      [
        set angle one-of [ 0 90 180 270 ]
        ifelse [ pcolor ] of patch-at-heading-and-distance angle 1 = blue and [ pcolor ] of patch-at-heading-and-distance angle 2 = blue
        and [ pcolor ] of patch-at-heading-and-distance angle 3 = blue and [ pcolor ] of patch-at-heading-and-distance angle 4 = blue
        [
          color_in
          set ship_length ship_length - 1
        ]
        [
          Spawn_a_boat_procedure
        ]
      ]
  ]
end
```

Because none of these solutions worked and with the deadline moving closer I decided to work on another section of code.

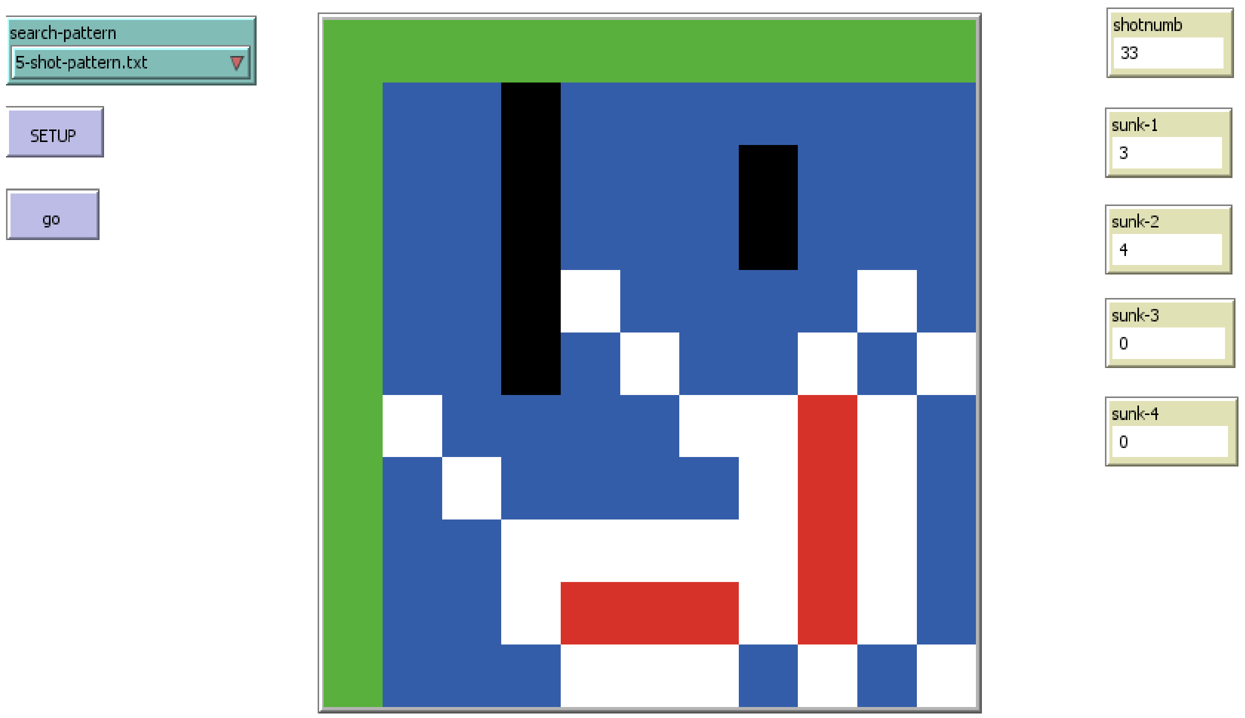
The next step in creating the program was to make a system so that when a boat was hit, it would seek out the remaining parts of the boat. This section was rather simple, for this I made a simple program that would check neighbors4 before coloring in each location depending on whether the patch was a hit or a miss. If the shot did not hit a boat then it was a miss so the patch would be colored white. If the shot did hit a boat then the patch would be colored red then rerun the code centered on the new patch. To verify that my program wasn’t negatively affecting my results I tested the random shot pattern against itself with one using the code to finish off the boat while the other would just shoot at random. This verified that my code was helping in the efficiency of my program.

Since I needed to be able to consistently place boats to move forward I put my focus back on spawning boats. I fixed the nobody issue by removing world wrapping and making the world an 11x11 with a border on the side of the world. The border would keep boats from wrapping around the world without the use of world wrapping which was causing the error.



Throughout the project because I was new to coding my mentor would show me different ways to solve problems by looking at other examples of code for Netlogo. During these meetings we analysed why these worked and compared what they did better as well as what I myself did better. After looking at these I was able to make much more optimised code by taking elements from their code and combining it with my own to make a better version of my program. By doing this I was able to figure out how to fix my nobody error.

After the main code was finished I had to figure out how to get the program to record the order of how boats were hit. This was a lot more difficult than I had expected. Due to the way I initially created the boats, I couldn’t just change a variable to have it read as destroyed. Because of this I had to rewrite most of the code that was used to create the boats. Instead of using a loop I had to create every boat individually so that they could all have a personal variable for their boat type. After that I created a section of code that would count the number of patches for that type of boat and if it hit 0 then the monitor would put the name of the boat that first got destroyed. While making this to count for the second, third and fourth destroyed ships however things got a bit more complicated. The first monitor would overwrite the other monitors leading to all monitors only showing a single boat name. To fix this I created a block of code that would check the fourth and third monitors before the first which let me change the first without overwriting the others. Here was the final result.


search-pattern: 5-shot-pattern.txt

shotnumb: 33

sunk-1: 3

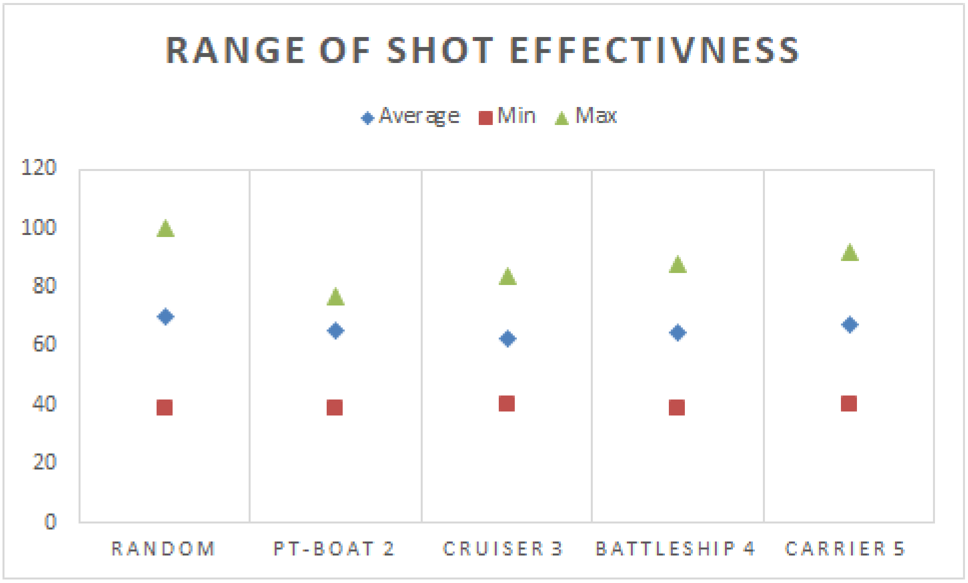
sunk-2: 4

sunk-3: 0

sunk-4: 0

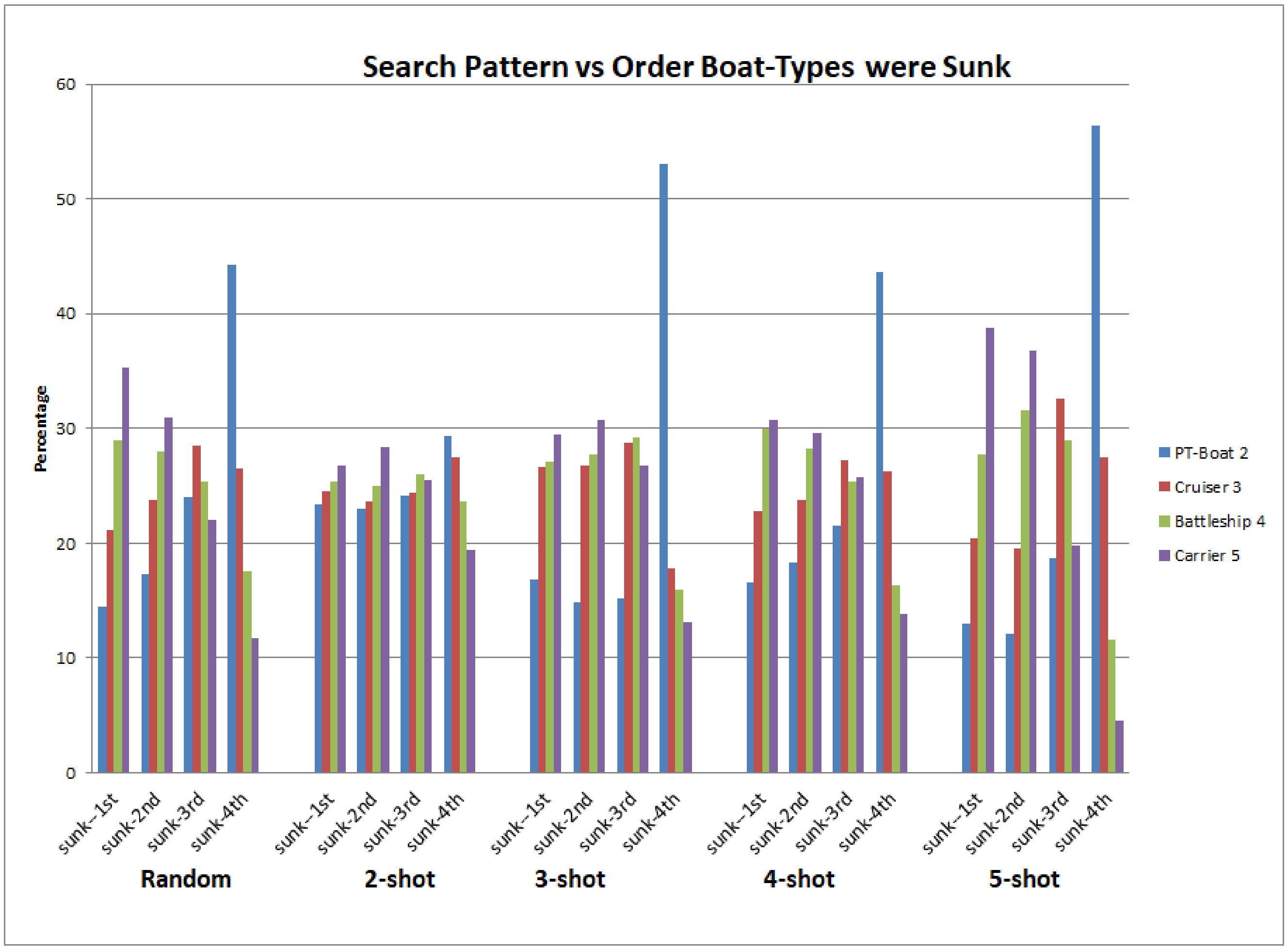
Results

Using the random shot as the baseline we can see how effective each pattern was in finding the boats. After running each pattern 10,000 it was shown that for the Skip-1 pattern it would take an average of 65 shots to find all of the boats. This is within the standard deviation of random so compared to random there is no major statistical difference in total shots fired. For the skip-2 pattern it took 63 shots to find all of the boats which are also within the standard deviation making skip-2 have no major difference in total shots fired. Skip-3 pattern followed this trend with it also taking on average 65 boats to find all of the boats. It's within the standard deviation and is not a significant enough of a difference. Finally the Skip-4 pattern required an average shot count of 67 which is also not a significant difference. Because all of the shot patterns had negligible differences in total shot count, my null hypothesis was correct.



Hypothesis #	Hypothesis Statement	Average Shots	Within SD1?	Supported?
H_0	Null- No Difference	Random = 70	All patterns	Yes
H_1	Skip-1 Pattern most effective	65	Yes	No
H_2	Skip-2 Pattern most effective	63	Yes	No
H_3	Skip-3 Pattern most effective	65	Yes	No
H_4	Skip-4 Pattern most effective	67	Yes	No

Even though the differences in shot count were negligible, the data showed that the order that each ship was found in differed heavily between each shot pattern. The program would find the equivalent sized boat more often than the other patterns. For example while the 4-skip pattern would find the carrier the most, it would find the battleship first nearly as often as it would find the carrier. This pattern was the same for all shot patterns besides the random.



As you can see, while the pt-boat is found last most often in all shot patterns it is found significantly more in the 1-skip pattern compared to the others. This is because the Skip-1 pattern cannot miss the pt-boat like the other patterns can.

The 2-skip pattern shows that despite being smaller than the other two boats, its chance of being hit is nearly as high as the battleship and carrier. It also shows a trend with the patterns where the boats smaller than the shot pattern get hit much less than the other boats. This trend continues with the skip-3 pattern as there is a significant drop in how often the cruiser is hit. For the 4-skip pattern this is shown even more as the carrier has a significantly higher chance of being hit before any other boat. This is because the larger the gap between shots the higher the chance that a boat can be missed and require the program to go back to fill in the gaps.

Significance of Results

While how you search patterns may not change the speed of which you find things it will change the order that you find things in. The biggest take away from this experiment is that by searching big you will find big and by searching small you will find small. This idea can be explored in many different ways whether it be in real life searching or by searching through data. A more refined version of this type of search could be useful in search or rescue as if a larger boat was lost at sea somewhere then it might be more useful to look in a large area instead of relentlessly searching. The same goes for searching for a smaller boat. It's better to search in a smaller area if the thing you're looking for is also smaller.

Reflection

There were a lot of things I had to learn in the making of this program. The first of which is the importance of pseudo-code when making a larger project like this. A lot of code had to be rewritten due to the lack of a planning phase. Another thing I learned was the importance of commenting what the code did as well as more detailed descriptions of complex code. I also learned the importance of meaningful variable names, especially when you have others looking at your code. Just because I understood it didn't mean that others or future me would understand what I was trying to do.

I also learned that things will not always turn out the way you expect them to. I was hoping to find the best way to win Battleship but instead found a new way of searching. While the discovery turned out to be interesting, it goes to show that there is always something hiding under the surface if you just take the time to look.

I would like to give my thanks to the many people who helped me during this project including my teacher Dr. Mueller, my mentors Mr, Edington and Mr, Henderson, Seungbin Chung for the idea of this project, and the kind teachers of the netlogo help sessions.

Citations:

Wilensky, U. (1999). NetLogo. http://ccl.northwestern.edu/netlogo/. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.