An Exploration of Discrete Mathematics Using Systems Based on Discrete Mathematics

Andrew Rindfleisch

17 December 2014

Introduction

This exploration asks the student to identify patterns in the output of wires whose values are manipulated by an "n by 1" matrix of cells. Each cell represents a Boolean function and manipulates its output wire based on its three input wires. Each cell may use a different Boolean function, or they may all use the same function.

Terminology

Before an in-depth discussion of Boolean Logic and Gating Networks can take place, one must first be armed with some basic terminology. First, a Boolean function is one that follows the form f : $\mathbf{B}^k \rightarrow \mathbf{B}$, where **B** = {0, 1} is the Boolean domain, or the domain containing only two values: true and false. There are 2^{2^n} Boolean functions of n degree, meaning that there are n inputs. For example, the number of Boolean functions of degree three (like those used in the exploration) is calculated 2^{2^3} = 256 possible functions.

A logic gate is an implementation of a Boolean function. It takes in one or more logical inputs and produces a logical output based on the function it is implementing. Common simple logic gates include AND, OR, NAND, NOR, and NOT gates. Logic gates may also implement Boolean functions that are more complex. For example, instead of simply checking to see if all three inputs are true (as an AND gate would do), a gate may check to see if the first and second inputs are true and the last input false ($xy\bar{z}$), or it may check if the first and third inputs are false or the first input true and the second input false ($\bar{x}\bar{z} + \bar{x}y$). A logical gate with three inputs could implement any one of the 256 different functions (as explained in the paragraph above) (objective #1).

Understanding

As with all other explorations, the first (and most difficult) task was to understand what exactly was being asked. Before effective problem solving can take place, one must thoroughly understand what the problem is and what resources are available to solve that problem. I have also found that it is helpful to look at effective solutions to similar problems to serve as a foundation for the solution.

I initially decided to just dive into the code and try to find a solution through programming (the reader will note that this is *not* an effective form of problem solving). I soon realized that I didn't understand the problem well enough to even start thinking about coding. I took a step back and broke the problem down into smaller problems. Essentially there were three parts to the problem: understanding how the cellular network is supposed to function, implementing a solution, and analyzing the output. Once I was able to understand how the network was supposed to function, I was then able to begin working towards a solution (objective #6).

A few hours of critical thinking later, I was able to obtain the proper output. I wrote a simple program to create a script that would run my program for all 256 Boolean functions and send the output to appropriately named files. Finding myself short of any slick pattern recognition software, I decided to brush up on my shell scripting skills and make my own resources. Instead of looking at each file independently, I decided to automate the process by using a script to diff all the files and find exact matches. The process was quick and dirty, but produced the results I

hoped would be helpful (objective #4). I grouped the identical outputs together and gave them a description. Below are my findings:

Pattern	# of Functions	Functions that exactly follow this pattern	Description of the Pattern
A	16	0- 7, 16-23	Just a star on top, rest of file is blank.
B	8	8- 11, 24-27	Line from top middle to bottom right (wraps to other side)
C	4	12-15	Almost identical to pattern B
D	4	28-31	Diagonal line to straight down (once it wraps to other side of screen)
E	16	32-39 and 48-55	Line straight down
F	2	40 and 41	Diagonal line to straight down (once it wraps); thickness of diagonal part alternates between 1 and 2 stars every other step.
G	2	42 and 43	Diagonal line to straight down (once it wraps); thickness of diagonal part is 2 stars.
H	2	56 and 57	Grows into a triangle, goes straight down after wrapping.
I	4	64-67	Large triangle grows from right side; left side is pattern of 1 and two stars; thin diagonal line precedes growing triangle.
J	4	68-71	Large triangle grows from right side; left side is pattern two stars; thin diagonal line precedes growing triangle.
K	4	76-79	Large triangle grows from right side; left side is four stars; thin diagonal line precedes growing triangle.
L	4	81, 83, 85, 87	Large triangles grow from right side and middle; left side is pattern of two stars
M	2	89, 91	Sierpinski triangles!
N	2	93, 95	About here I thought about why I was doing this.
O	2	96 and 97	Still thinking…
P	2	98 and 99	More thinking went on here.
Q	2	209 and 213	At this point I realized what is explained in the following paragraph…
R	4	232, 234, 236, 238	
S	4	248, 250, 252, 254	
T	4	249, 251, 253, 255	
U	164	44-47, 58-63, 72-75, 80, 82, 84, 86, 88, 90, 92, 94, 100-208, 210-212, 214-231, 233, 235, 237, 239-247	Unique pattern (doesn't *exactly* match the output of any other function).

Analysis

As I sat there in my chair looking at my superficial analysis, I realized that it did not account for any backwards symmetry nor did it check for similarity between outputs that were not exactly symmetrical. I was unsatisfied by this approach and changed my methods. I instead decided to make one large file that contained all 256 outputs in succession so that I could view them all at once and compare them. This made it much easier to look at all the outputs and quickly analyze the different patterns (objective #3). Some of my favorite patterns I found are included below:

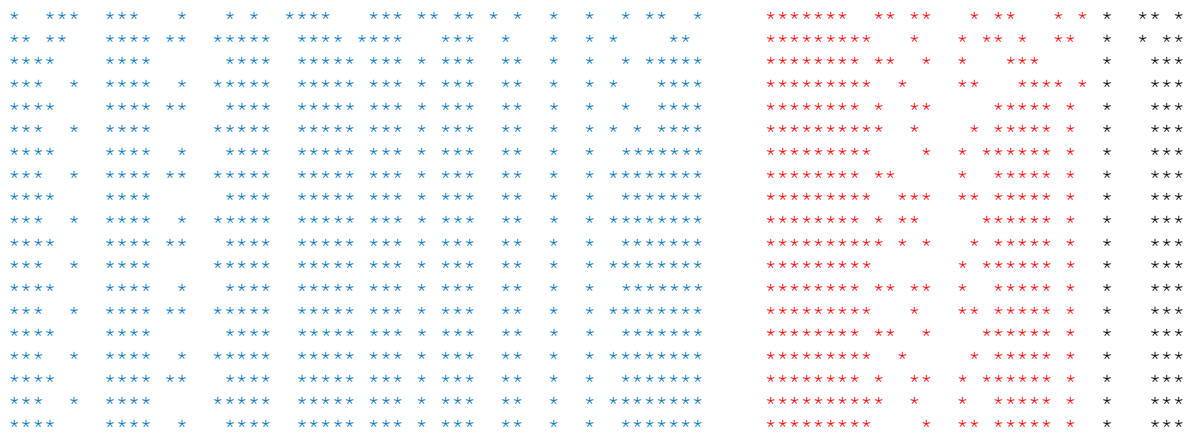
Function Number	Description of the Pattern
215	"Stars and Stripes"; a patriotic pattern reminiscent of the star spangled banner.
90	Generates a Sierpinski triangle
30	"O Christmas Trees"; a festive pattern reminiscent of two Christmas trees.
57	"The Great Divide"; forms a triangle whose right half is twice as populated with stars than is its left half.
60	"Sierpinski's leaning tower"; very similar to number 90, but only grows to the right which gives it a "leaning" effect.
129	Creates an inverse Sierpinski triangle where the stars and spaces are reversed.
133	"Cathedral Spires"; looks like a close up of a silhouette of the spires on a cathedral or the Salt Lake Temple.

After looking at these patterns, I noticed that many of them share similarities. For example, some patterns mirror each other from left to right (see patterns 60 and 110), and some patterns are inverses of each other, meaning the position of the stars and blanks have been swapped (see patterns 110 and 153).

At the urging of Brother Neff, I researched different cellular automata, of which this exploration is an example. Specifically, I learned that this exploration is an elementary cellular automaton. What I have referred to as pattern numbers are formally called rules. The relations that I called

mirrors are mirror rules, and the relations that I called inverses are called complementary rules. The outputs of an elementary cellular automaton can be used to model different discrete applications, such as generating functions.

It was interesting to note that by seeding each cell with a random Boolean function, the output seems to partition itself into separate columns. Each column repeats its own pattern, and each pattern generally has a very small length (usually lasting only two or three generations before repeating, though it may be much longer). What is most fascinating to me is that after about 10 iterations, any interaction between the columns ceases and they continue downwards repeating the same pattern, reaching total stability. I initially conjectured that seeding each cell with a different function would produce chaos, but the complete opposite occurs. Below is a typical example of a "randomized" pattern. Notice how the columns on the left side (marked in blue) have a very short period in their patters, while the columns on the right side (marked in red) are still in the process of stabilizing (and perhaps never will fully stabilize!):



Optimization

The outcome of this exploration is not to simply obtain the desired output. For me, I wanted this exploration to be one of optimization. I wanted to use all my skills as a programmer and a mathematician to create a better work than I could have before taking this class. I was able to look back at previous explorations for inspiration (as prompted by Brother Neff). As I look at the option to run all 256 Boolean functions, I can't help but think of my dear old friend ∀, the universal quantifier (objective #2). The option to run all Boolean functions could be expressed as $\forall f\big(C(f)\big)$, where f represents a Boolean function of degree 3 and $C(f)$ = "run the program using function f". I also decided to use the System class created by Brother Neff that we used in exploration 2 to make input easier as I tested my program (objective #5).

Conclusion

As I walk away from this exploration, I feel satisfied. I have thought a lot about this exploration and previous ones. In doing so I have realized that each exploration we have completed uses discrete mathematics through the medium of computer science to model different principles of discrete mathematics. Yes, you read that correctly. We are using discrete math to model discrete math. Loops, Boolean functions, and graphs are simply implementations of but a few of the many branches that compose the grand tree of discrete mathematics. This exploration has reinforced in my mind that discrete mathematics applies to all parts of computer science (objective #8). However, I would err in stopping there and concluding that its applications *only* extend at computer science. Nay, whether it be figuring out the odds that a randomly selected sock chosen from a laundry bin will be green or determining which road to plow by creating a

graph with weighted edges and using Kruskal's (or perhaps Prim's!) algorithm to determine the best path, the principles of discrete mathematics are inescapable (objective #7)!