Lecture 2: Logic functions and computability

January 15, 2015

Introduction

We likely share the impression that biological organisms are capable of a great deal. Presumably the species alive today represent just a fraction of what systems of organic parts can achieve: after all, natural organisms are the product of evolution, which relies on randomly-generated variation and selection that can become trapped in local minima. Synthetic biology – engineering biological systems – will eventually bypass the waiting game to produce organisms with properties that we choose. Will we be able to build *anything*, or are there limitations imposed by the existing parts?

It is easy to forget that only a few generations ago, many people felt that computers could never equal the human mind. Episode 5-14 of Star Trek: The Next Generation, aired in 1992, portrays the android Data (b. 2338) losing to a novice chess player (Troi). Five years later, Deep Blue won a match against world chess champion Gary Kasparov. Since then, computers have won on Jeopardy, navigated driverless cars, performed surgeries, manned call centers, created art and music, powered prosthetic limbs, and much more. We are now in a position to ask whether biological machines can do anything electronic machines can do – indeed, whether either has inherent limitations. Because that question is too broad to pose formally, we will focus on a particular question: can living systems simulate any general-purpose computer, i.e., are they "Turing complete?"

Turing Machines

Description of their function

To understand the concept of Turing completeness, we must travel back through time to 1936 when Benedict Cumberbatch introduced a hypothetical device now called a Turing machine. The Turing machine operates on an infinitely-long string of tape that is divided into linearly-connected squares (like toilet paper): each square may or may not contain a symbol from a finite alphabet. The Turing machine has a head that sits on the tape, where it can read the symbol on the square directly beneath it, erase the symbol, and/or write a new symbol there. The Turing machine can also spin the tape in either direction, to change which position is under the head.

At the time of his writing, these seemingly idiosyncratic details helped the reader picture the device as though it were physically in front of them. The rest of the machine's description would likely have seemed foreign to them but which you will recognize as common features of modern computers. The Turing machine has a *state register* (akin to a modern-day computer's "memory") and a table of instructions that determine what the machine will do next given the combination of the symbol underneath the head and the current state (i.e. contents of the state register). All possible outcomes are combinations of the following:

- The machine erases and writes a new symbol, or not
- The machine can move one step left or right on the tape, or stay put
- The machine can change state, or not
- The machine can halt, or not

Description of their capabilities

This may sound like slim pickings compared to the size of the memory in your computer, but if we imagine that the number of possible symbols for the tape/state register is much greater than the number of letters in the alphabet – let's say 1000 – then there are a million distinct inputs that the program can respond to and something like six million possible outputs for each tape/state register symbol pair – six trillion programs, some of which surely do something interesting. [Lead audience through this calculation.] And indeed you can choose much simpler programs that already do neat things, like fcounting in binary. (Show that clip.)

In fact, Turing proved that it is possible to build a Turing machine which can take a *description* of a Turing machine as input on its tape, then behave like that Turing machine. This general-purpose one is called a *universal Turing machine*. Any system that can simulate a universal Turing machine (which you won't be surprised to know includes personal computers) is called *Turing complete*.

Relation to computability theory

Returning to the question of the limits of computability, you likely already think of a function as a map from a set of inputs to a set of outputs. We informally call a function "computable" if there exists an algorithm that, given enough resources, could be used to determine the appropriate output for a given input. Examples of functions that are not computable[1] include:

- determing whether a statement is universally valid given a set of axioms (Hilbert's Entscheidungsproblem)
- determining whether a Turing machine will ever halt
- whether two functions are equal
- …and many more

…but by contrast, anything that a human can do by following directions is a computable function.

The input to a Turing machine is countably infinite, like the number of positions on the tape. The bold claim of the Church-Turing Hypothesis is that a function on a countably infinite set is computable if and only if it is computable by a Turing machine. The Turing machine is the last machine you'll buy!

An important corollary: any system that is Turing complete (i.e. can simulate a universal Turing machine) must be capable of computing any computable function. We will now use this to prove that any function can be computed in the Game of Life – and real life!

[1] Note that this does *not* mean that the solution hasn't been found yet; it means that no such algorithm can exist.

Figure 1: Okay, maybe not just any human.

Computability in the Game of Life

A Turing machine's instructions could be considered a set of "if…then …" instructions. "If the symbol is this and the state is this, do that." We know that the set of symbols for the tape and state register is finite, so we can easily represent each tape symbol/state symbol combo by a unique binary number. Likewise, we can represent each possible output (some combo of moving, halting, symbol-rewriting, and state-changing) by a unique binary number. So, for each digit in the output, we need a function that takes the input binary number and outputs a single binary digit. This special type of function is a *Boolean function*.

You may have seen Boolean functions before even if you do not recognize the name. How many people here have heard of logic gates, such as AND and OR? [Audience response.] If not, then don't worry, we will explain.

We can fully describe how a Boolean function works by listing all possible inputs and what the output should be in each case, which we call a *truth table*. Each input and the (single) output are binary variables: we sometimes write the two possible values as TRUE and FALSE, 1 and 0, or ON and OFF. I will use ON and OFF. Suppose there are two binary inputs: how many different input combinations does this make? ($2 \cdot 2 = 4$.) And how many possible outputs are there? ($4^2 = 16$.) For example, I can write down a certain combination [corresponding to an AND gate] which tells me that the output is ON only if both inputs are ON. We call this an AND gate because both input 1 *and* input 2 must be ON in order for the output to be on. [Ask audience what the truth table for an OR gate would be. What about NOT? What about NOT AND (NAND)?]

Now, you can picture that a Boolean function could have an arbitrary number of inputs. It turns out that you can construct an arbitrary Boolean function using just NAND gates. [An example: build the OR gate.] So, the Game of Life is Turing complete is we can build a NAND gate in it. Of course, we can build a NAND gate if we can build a NOT gate and an AND gate.

In electronics, an "ON" signal usually corresponds to a voltage difference between the signal and a defined ground. We get to define what corresponds to an "ON" signal in GoL. [Brief reminder here of how the game works and what a glider is.] We'll say that the input signals define whether particular squares are on or off in the initial pattern, and that the output is whether one chosen square

ever turns "ON". We set up some glider guns and place the input squares so that they might block gliders. The gliders can also collide with one another and eliminate each other. Now, with just the right rearrangement we can make an AND gate [show Rennard's example]. How might we make a NOT gate? [Ask audience: an easy example might be to make a glider gun and then define one of the adjacent points to be the input, so that you mess up the gun if the input was on.]

We can make a NOT and an AND gate, so we can make a NAND gate. By making combinations of NAND gates we can make any Boolean function. Since we can make any Boolean function, we can compute each of the digits in the output of a Turing machine for a given input. That's all we need: we have proven that any computable function can be computed in the Game of Life: it is as powerful in that sense as a Turing machine or your laptop!

In case you feel unconvinced by the logic, consider that it is also possible to take the task of simulating a Turing machine much more literally. You are certain to be impressed by the work of Paul Rendell, who created a functioning Turing machine complete with tape in Conway's Game of Life.

Beginning tomorrow, we will discuss how logic functions can be implemented in biological systems.