If that Was True I Would Know it!
A Result in Kolmogorov Complexity

(This manuscript is essentially a chapter of *Problems with a Point.*)

1 Point

As a graduate student, in 1983, I came across a book in the library that claimed to solve Fermat's Last Theorem (FLT) using elementary methods. (This was before Wiles showed FLT true using rather advanced methods.) I **knew** that the claim was false. Why? Because **if FLT had really been solved, then I would know that.** I was not bragging about how much math I knew; I was bragging about how much math I knew *of.* Was that a rigorous proof technique? No, but it is useful and often correct.

I now tell the tale of how I used this technique in Kolmogorov Complexity. In particular, I **knew** that the Kolmogorov function could not be of intermediary Turing degree because **if there were a natural intermediary Turing degree then I would know that**. This chapter will be self-contained; hence, I will define all of these terms. For more on Kolmogorov complexity see the classic book by Li and Vitanyi [Li and Vitányi(2008)].

2 How to Measure Randomness Intuitively

Intuitively the string 000000000000000000000 does not seem random. How to make this rigorous? Note that there is a program of length $\lg n + O(1)$ that prints out 0^n:

```
for $i = 1$ to $n$ print(0)
```

Conversely, the string 0110100011000000111010101010001100 does seem random. The shortest program to print it out might be

```
print(0110100011000000111010101010001100)
```

which is roughly the length of the string itself.

Taking a cue from the above two examples, we will define the *randomness of a string* x to be the size of the shortest program that prints x. We need a formal notion of program.

3 A Short Introduction to Computability Theory

We will not define Turing machines formally. All you need to know is the following:

- A function can be computed by a Turing machine iff the function can be computed by a computer program, say, written in JAVA.
- Much like Java programs, it is quite possible that a Turing machine run on an input will never halt.
- Turing machines have a finite description, which can be interpreted as code. That is, given a description M and an input x, one can run $M(x)$.
- The program

  ```
  For $i = 1$ to $n$ print(0)
  ```

  has length $\lg n + O(1)$.
- If x is a string of length n then the program

  ```
  print($x$)
  ```

  has length $n + O(1)$.

Notation 3.1 $M_1, M_2, \ldots$ is the list of all Turing machines. Let $i, x \in \mathbb{N}$. If, when M_i is run on x, the computation terminates with output y, then we write $M_i(x) = y$. We take the index i to be in base 2 so the length of M_i is $|i| = \lg(i) + O(1)$.

Def 3.2 A function $f : \mathbb{N} \to \mathbb{N}$ is *computable* if there exists a Turing machine M such that, for all x, $M(x) = f(x)$. A set A is *decidable* if its characteristic function is computable.

Example 3.3 Most functions you encounter in mathematics are computable. For example, addition, squaring, exponentiation, and given n find the nth prime. Most sets you encounter in mathematics are decidable. For example, evens, primes, and squares.

Def 3.4 Let $i, s, x \in \mathbb{N}$.

$$M_{i,s}(x) = \begin{cases} y & \text{if the computation } M_i(x) \text{ halts within } s \text{ steps and outputs } y \\ \text{NO} & \text{otherwise} \end{cases} \tag{1}$$

The function $f(i, s, x) = M_{i,s}(x)$ is computable: Run $M_i(x)$ for s steps and either (1) it halts within s steps, so output what it outputs, or (2) it does not, so output NO.

We now give an example of an undecidable set.

Def 3.5 HALT is the set $\{x : M_x(x) \text{ halts }\}$.

Note 3.6 A more natural set for HALT would be $\{(x, y) : M_x(y) \text{ halts }\}$. However, the version we use is easier to work with and ends up being equivalent, as we will see later.

Theorem 3.7 HALT *is undecidable.*

Proof: Assume, by way of contradiction, that HALT is decidable. Hence we will be able to make the query $x \in$ HALT in our program and get an answer back. Let M_i be the Turing machine that does the following

1. Input(x)
2. Ask $x \in$ HALT. Note that this will be YES if $M_x(x)$ halts and NO if $M_x(x)$ does not halt.
3. If $x \in$ HALT then go into an infinite loop, else halt.

Is $i \in$ HALT? We show that both answers, YES and NO, lead to a contradiction.

If $i \in$ HALT then $M_i(i)$ halts. Hence $i \in$ HALT. When M_i is run on i, the instructions in step 3 say that it goes into an infinite loop. Hence $i \notin$ HALT. This is a contradiction.

If $i \notin$ HALT then $M_i(i)$ does not halt. Hence $i \notin$ HALT. When M_i is run on i, the instructions in step 3 say that the computation halts. Hence $i \notin$ HALT. This is a contradiction.

▮

Geoffrey Pullum [Pullum(2004)] presents the proof that Halt is undecidable as a poem in the style of Dr. Seuss.

We need the notion of *If I had access to the function g then the function f would be computable.* We could define oracle Turing machines formally but instead we define it informally.

Def 3.8 An *Oracle Turing Machine* is a Turing machine that has the ability to make a call to an unspecified function g, called *the oracle.* The Oracle Turing machine is defined independent of the function. We denote an oracle Turing machine by $M_i^{()}$. The notation $M_i^g(x)$ means that we run the $M_i^{()}$ with oracle g and on input x.

Def 3.9 Let $f, g : \mathbb{N} \rightarrow \mathbb{N}$. We say $f \le_{\rm T} g$ if there is an oracle Turing machine $M^{()}$ such that M^g computes f. This is called a *Turing reduction* We say $f <_{\rm T} g$ if $f \le_{\rm T} g$ but $g \not\le_{\rm T} f$, and $f \equiv_T g$ if $f \le_{\rm T} g$ and $g \le_{\rm T} f$.

Exercise Let $\mathrm{HALT}_0 = \{(x, y) \mid M_x(y) \text{ halts }\}$. Show that HALT $\le_{\rm T}$ HALT_0 and $\mathrm{HALT}_0 \le_{\rm T}$ HALT. This is what we meant in the note following Definition 3.6 by saying that the two sets were equivalent.

The following lemma we leave to the reader:

Lemma 3.10 *If $f \le_{\rm T} g$ and g is computable then f is computable.*

Given a function how do you prove that is undecidable?

One way is by contradiction. The proof that HALT is undecidable is by contradiction. We will later define the Kolmogorov function and show it is not computable by contradiction.

The most common way to show that a function f is not computable is to take a known undecidable set A (almost always HALT) and show $A \le_{\rm T} f$.

4 Intermediary Sets

We can rewrite the statement HALT *is undecidable* as $\emptyset <_{\rm T}$ HALT. A set A such that $\emptyset <_{\rm T} A <_{\rm T}$ HALT is called an *intermediary sets.* In 1950 it was an open problem to determine if there were any such sets. In 1954 Post and Kleene first showed there were such sets. What are these sets? I once had the following conversation with my Darling about these sets:

Bill: Darling, if I told you there were sets that were not decidable but easier than the halting problem, would you find that interesting?

Darling: Yes, unless…

Bill: Great! Because there are. Oh, I interrupted you – what were you saying.

Darling: I would find the existence of such sets interesting unless they were some dumb-ass sets that people constructed *just* for the sole point of being undecidable but easier than the halting problem.

Bill: You nailed it!

In a nutshell, such sets are not natural. Hence the sets themselves are not interesting. It would be nice to have natural, and hence interesting, intermediary sets.

Most computability theorists think that there are no natural intermediary sets. It is not clear how to prove this. In fact, it is not even clear how to state this rigorously.

5 How to Measure Randomness Formally

We use TM to abbreviate "Turing machine."

Def 5.1 The *Kolmogorov complexity of a string x*, denoted $C(x)$, is the length of the shortest TM M such that $M(0) = x$. We often call a TM that prints out a string x a *description of x*.

Note 5.2 The definition of C depends on the formal definition of TM. t is possible that if you define TM's using Java programs then $C(0^n) = \lfloor \lg(n) \rfloor + 1000$ whereas if you define TM's using Fortran programs then $C(0^n) = \lfloor \lg(n) \rfloor + 5$. However, for any two types of TMs, there is a constant c such that the two $C(x)$'s differ by at most c. Hence this will not affect any of our results.

C is a measure of randomness. If $C(x) \geq |x|$ then we think of x as being random. Are there such strings? Yes!

Lemma 5.3 *For all $n \geq 1$ there is a string $x \in \{0,1\}^n$ such that $C(x) \geq n$.*

Proof: Assume, by way of contradiction, for all $x \in \{0,1\}^n$, $C(x) < n$. Map each $x \in \{0,1\}^n$ to the program that prints it. Note that this map is 1-1. There are 2^n elements in the domain and

$$\sum_{i=0}^{n-1} 2^i = 2^n - 1$$

in the range. Hence the map cannot be 1-1. Contradiction. ▮

How hard is C? We first show that $C \le_{\mathrm{T}}$ HALT and then that C is not computable.

Theorem 5.4 $C \le_{\mathrm{T}}$ HALT.

Proof:
Let c be the constant such that, for all x, $C(x) \le |x| + c$.

1. Input x. We want to know $C(x)$.
2. Let $M_0, M_1, \ldots$ list out all TM's in order of size (within the same size list it out lexicographically).
3. For $e = 0$ until you get a YES answer: ask the HALT oracle: *Does $M_e(0)$ halt and output x?*
4. The first time you get a YES, halt and output $|M_e|$.

▮

We need a lemma

Theorem 5.5 *C is not computable.*

Proof:
Assume, by way of contradiction, that C is computable. Assume also that the program for C is of size n. Consider the following program

```
for each $x \in \{0,1\}^{2n}$
    compute $C(x)$
    if $C(x) \ge 2n$ then print($x$) and stop.
```

By Lemma 5.3 there will be an $x \in \{0,1\}^{2n}$ with $C(x) \ge 2n$. Hence such and x will be found and output. Hence the above program *prints out a string x such that $C(x) \ge 2n$.*

This program is of size $n + \lg(n) + O(1)$. Hence $C(x) \le n + \lg(n) + O(1)$.

OH, so we have

$$2n < n + \lg(n) + O(1)$$

This is a contradiction.

More succintly: we found a string of HIGH Kolm-complexity that has a SHORT program to print it. ▮

6 Is the Kolmogorov Function Intermediary?

As a grad student, in 1984, I came across Theorems 5.4 and 5.5 which together imply $\emptyset <_{\mathrm{T}} C \le_{\mathrm{T}}$ HALT. Unlike most proofs of non-computability, the proof that C was not computable *did not* show $HALT \le_{\mathrm{T}} C$. Hence it was possible that C is an intermediary set. I **knew** that possibility was false. Why? Because **If there were a natural intermediary set then I would know that.** I was not bragging about how much computability theory I knew; I was bragging about how much computability theory I knew *of.* Is that a rigorous proof technique? No, but it is useful and often correct.

Was I correct? Yes. I asked around (this was before the Web and even email was not as common in 1984 as it is in 2018) and eventually Peter Gacs (a professor at Boston University) gave a proof that HALT $\le_{\mathrm{T}} C$ (on paper) to Mihai Gereb (a graduate student at Harvard) who gave it to me. I would have preferred to be wrong and to have seen a natural intermediary set. Oh, well.

Here is the proof:

Def 6.1 Let $C_s(x)$ be the size of the shortest TM M such that $M(0) = x$ and halts within s steps.

Theorem 6.2 HALT $\le_{\mathrm{T}} C$.

Proof:

Here is the algorithm for HALT that uses C as an oracle.

1. Input(x) (we want to know if $M_x(x)$ halts). Let $|x| = n$.
2. Find s_0 such that, for all $y \in \{0,1\}^{2n}$, $C_{s_0}(y) = C(y)$. (This step uses the oracle for C.)
3. Run $M_x(x)$ for s_0 steps. If it halts, then output YES. If not, then output NO. (We still need to prove that this is correct.)

We need to show that if $M_x(x)$ does not halt within s_0 steps then it never halts. Assume, by way of contradiction, that $M_x(x)$ halts in $s \geq s_0$ steps. Note that, for all $y \in \{0,1\}^{2n}$, $C_s(y) = C(y)$. The following algorithm will be a short description of a string that lacks a short description.

1. Run $M_x(x)$. Let s be the number of steps it took to halt.
2. For all $y \in \{0,1\}^{an}$ compute $C_s(y)$ (which is $C(y)$).
3. Let y be a string of length an such that $C_s(y) \geq |y|$.
4. Output y.

The above algorithm can be described with $n + O(1)$ bits. Hence

$$C(y) \leq n + O(1).$$

By the definition of s we have

$$C(y) = C_s(y) \geq |y| = 2n.$$

Hence

$$2n = |y| \leq C(y) \leq n + O(1)$$

This yields a contradiction.

▮

7 The Point Reiterated

Let Q be a statement in mathematics that is either true or false. You don't know which. Should you try to prove Q or $\neg Q$? How do you decide what to spend more time on? You can do examples or see what fits in with other mathematics. You can see what people who have worked on the problem think. And you can sometimes use:

If Q is true then I would know that. Hence I will try to prove $\neg Q$.

I am not claiming this as a rigorous proof technique. But it may be used as a starting point. Note that I might have wasted a lot of time trying to understand an allegedly easy proof of FLT or trying to show that C was intermediary had I not employed this method.

References

[Li and Vitányi(2008)] Li and Vitányi (2008). *An introduction to Kolmogorov complexity and its applications* (Springer, New York, Heidelberg, Berlin), this is the third edition.

[Pullum(2004)] Pullum, G. (2004). Scooping the loop snooper, `http://www.lel.ed.ac.uk/~gpullum/loopsnoop.html`.