CS 30: Discrete Math in CS (Winter 2020): Lecture 30

Date: 4th March, 2020 (Wednesday)

Topic: Infinite Sets: Uncountability and Undecidability

Disclaimer: These notes have not gone through scrutiny and in all probability contain errors. Please discuss in Piazza/email errors to deeparnab@dartmouth.edu

---

1. **Diagonalization and Uncountability.** Not all sets are countable. Indeed:

Theorem 1. The set of boolean function $\mathcal{F}$ is *not* countable.

Proof. Recall, $\mathcal{F}$ is the set of all Boolean functions $f : \mathbb{N} \to \{0,1\}$. Suppose, for the sake of contradiction, $\mathcal{F}$ was indeed countable. That is there is an injective function which maps every $f \in \mathcal{F}$ to a natural number in $\mathbb{N}$.

Using the ordering algorithm from last lecture, this means we can take all the functions in $\mathcal{F}$, and line them in some order $(f_1, f_2, f_3, \ldots)$. More precisely, given any $n \in \mathbb{N}$, there is a mapping which gives us the nth function $f_n \in \mathcal{F}$ (this is the inverse mapping which maps functions in $\mathcal{F}$ to $\mathbb{N}$.)

We now construct a table $T : \mathbb{N} \times \mathbb{N}$ where the nth row consists of the function $f_n \in \mathcal{F}$, where the mth column of the nth row is simply the evaluation $f_n(m)$. This table is key to the proof.

Consider the *diagonal* of this table. Note that for *any* $n \in \mathbb{N}$, the entry $T[n,n]$ is precisely $f_n(n)$; the evaluation of the nth function on the number n. Now consider the function $f^* : \mathbb{N} \to \{0,1\}$ defined as

$$\forall n \in \mathbb{N}: \quad f^*(n) := 1 - T[n,n] = 1 - f_n(n) \tag{1}$$

Since $f^* \in \mathcal{F}$, there must exist some natural number $N \in \mathbb{N}$ such that $f^* = f_N$. That it, $f^*(n) = f_N(n)$ for all $n \in \mathbb{N}$. In particular, $f^*(N) = f_N(N)$. But this contradicts (1) for $n = N$. Therefore, $\mathcal{F}$ is not countable. □

Remark: *This is one of my all time favorite theorems.*

As discussed last time, as a corollary we get

Theorem 2. There must be uncomputable Boolean functions.

Historically, George Cantor, the discoverer of this diagonalization trick, used it to prove the uncountability of reals. I sketch it below. It is almost exactly the same as the one above.

Theorem 3. The set $[0,1]$ is *not* countable.

Proof. We prove this by contradiction. Suppose, for the sake of contradiction, the set of reals $[0, 1]$ was indeed countable. Then, by the definition of countability and the algorithm to order countable sets, we can order *all* the elements of $[0, 1]$ as $(r_1, r_2, r_3, \ldots)$. More precisely,

$$\text{For any } s \in [0, 1], \text{ there exists some natural number } k \text{ such that } s = r_k. \tag{2}$$

We now construct a table $T : \mathbb{N} \times \mathbb{N}$ where the nth row consists of the decimal representation of the real r_n (with perhaps an infinite padding of 0's.). This table is key to the proof.

Consider the *diagonal* of this table. Note that for *any* $k \in \mathbb{N}$, the digit in the position $T[k, k]$ is the kth digit after the "binary point" in the binary representation of the number r_k.

Now construct an *infinite string* s^* as follows: for every $k \in \mathbb{N}$, define $s^*[k] := (T[k, k] + 1) \bmod 10$[1]. Now consider the real number r formed by taking $0.s^*$, that is, s^* after the decimal place. Note that this real number r lies in $[0, 1]$. Therefore, there must exist some ℓ such that $r_\ell = r$.

In particular, the ℓth bit after the decimal place in r must be $T[\ell, \ell]$. However, *by construction*, the ℓth bit after the decimal place in r is $s^*[\ell] \neq T[\ell, \ell]$. Contradiction. □

2. **The Halting Problem.**

We now describe a *concrete* decision problem for which there cannot exist any algorithm /procedure /Python code. This theorem is due to Alan Turing. The problem is called the Halting Problem.

Input: The input to the problem is two *strings* A and I. The first string is to be interpreted as a piece of code (Python code, say). The second string is to be interpreted as an *input* to this code.

Output: The problem is to determine whether $A(I)$ halts in a *finite* amount of time.

Remark: *Note that A and I could be garbage strings. That is, A is perhaps not a valid code. In which case $A(I)$ halts (it gives an error; but it halts). Or I may not be in the correct format (the code expected integers and you gave it a float). But these cases are "easy" for the Halting Problem: it can perhaps figure out easily that $A(I)$ halts.*

Theorem 4. There is no finite time procedure which for *every* input A and I decides *correctly* whether or not $A(I)$ halts.

That is, the Halting Problem is *undecidable*.

Proof. Suppose, for the sake of contradiction, that there indeed existed a finite time procedure $H(\cdot, \cdot)$ which could take two strings A and I, and correctly figure out whether $A(I)$ halts or not.

Consider the following algorithm which takes input a string B and acts as follows:

[1] In fact any setting **other than** $T[k, k]$ would work.

L1. First the algorithm runs the subroutine $H(B, B)$. That is, it asks the procedure $H(\cdot, \cdot)$ whether the string B, when interpreted as a Python program, would halt when run on the *same* string B interpreted as data. By our supposition, in finite time, $H(B, B)$ would return YES or NO.

L2. If $H(B, B)$ returns YES: we run an infinite loop.

L3. If $H(B, B)$ returns NO: we return 1 and halt.

The above algorithm can be coded into a Python program. Let this bit of code be called C. Note that C is also a string. And therefore $H(C, C)$ must be return a value – YES or NO. Now we will reach a contradiction.

Case 1: $H(C, C)$ return YES. That is, the procedure H looks at the string C and decides that $C(C)$ *halts*. But now consider the run of $C(C)$ by looking at the code C. In Line L1, the code runs $H(C, C)$. Since we have assumed in this case that $H(C, C)$ is YES, the algorithm branches to Line L2. However in Line L2, the run goes into an infinite loop. That is, $C(C)$ *never halts*. In this case, we have reached a contradiction.

Case 2: Perhaps, $H(C, C)$ returns NO. That is, the procedure H looks at the string C and decides that $C(C)$ *never halts*. But now consider the run of $C(C)$ by looking at the code. In Line L1, the code runs $H(C, C)$. Since we have assumed in this case that $H(C, C)$ is NO, the algorithm branches to Line L3. However in Line L3, the run returns 1 and halts. That is, $C(C)$ *halts*. Again, we have reached a contradiction.

Therefore, in all possible scenarios (there are only two: YES or NO), we have reached a contradiction. We are forced to conclude that our supposition must be wrong. That is, there cannot be a finite time procedure $H(\cdot, \cdot)$ which for every A and I passed to it can decide whether $A(I)$ halts or goes into infinite loop. □

Remark: *The above proof is really the same diagonalization trick that we saw for the uncountability of reals. To see this, note that the set of strings is countable. Suppose the strings are named $(S_1, S_2, S_3, \ldots)$. Think of constructing the table $T[\mathbb{N} \times \mathbb{N}]$ where $T[S_i, S_j]$ contains YES if $H(S_i, S_j)$ returns YES, and contains NO otherwise.*

Now consider the diagonal of this table. This contains the answers to what $H(S_i, S_i)$ returns for all natural numbers i. Now construct the algorithm which takes input B, sees what $H(B, B)$ is, and does the opposite (that is halts if $H(B, B)$ doesn't and vice-versa). Just like we did for the case of reals. This is a finite algorithm and can be encoded as a string (a python code). Since it is a string, it is some S_k for some number k in the ordering.

The contradiction is obtained by looking at $H(S_k, S_k)$. If this YES, then S_k should halt on S_k, but S_k does the opposite. *Same for vice-versa.*

Remark: *Although the Halting Problem was the "first" undecidable problem, there are other decision problems which are also undecidable. Another one, quite different from the Halting Problem is this one:*

Given a polynomial *equation on* integer *coefficients and a finite number of un-*

knowns, is there a solution which takes integer *values?*

For instance, if the polynomial is $5x + 9y = 3$*, then indeed there are many solutions one being* $x = -3, y = 2$*. Or suppose the polynomial is* $3x^2 - 5xy + y^3 = 5$*, then it does have a solution* $x = 3, y = 2$*. On the other hand, a polynomial equation like* $x^2 + y^2 = 3$ *doesn't have any integer solutions.*

The question is then to decide whether there is a finite algorithm which takes as input such a polynomial equation and says YES if there is an integer solution, and says NO otherwise. This problem is called Hilbert's 10th problem, and was proved to be undecidable in 1970.