CS 30: Discrete Math in CS (Winter 2019): Lecture 6

Date: 11th January, 2019 (Friday)
Topic: 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 Reals are Uncountable

Although the rationals are countable, the set of reals are not. This beautiful proof was discovered by Georg Cantor and the trick is called diagonalization.

- **The Binary Point in the Binary Notation.**

  First let us recall the binary notation. Any real number $r \in (0,1]$ can be expressed as an infinite string of 0s and 1s following the binary point. We are not going to prove this here, but we recalled the procedure in class for finding binary notations for rational numbers.

  For instance, $1/4$ is $0.0100000\ldots$ and $5/8$ is $0.101000\ldots$, and $1/3 = 0.0101010101010101\ldots$. ✍

  > **Exercise:** *What is the binary expression for* $1/7$*?*

  Furthermore, given any infinite string s of 0s and 1s, it represents the real number using the following map.

  $$s \mapsto \sum_{n=1}^{\infty} s[n] \cdot \frac{1}{2^n} \tag{1}$$

  ✍

  > **Exercise:** *If* s *is not the all* 0*s string, show that* (1) *maps* s *to a real number in* $(0,1]$*.*

  ✍

  > **Exercise:** *Given two* different *infinite strings* s *and* s' *of* 0*s and* 1*s, why are the two real numbers represented by* (1) different*.*

- **Diagonalization.**

  > **Theorem 1.** The set $(0,1]$ is *not* countable.

  Proof. Suppose not, that is, suppose $(0,1]$ is 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 binary representation of the real r_n (with perhaps an infinite padding of 0's.). This table is key to the proof. So it is instructive to pick 6 or 7 real numbers and actually writing out the first 6 or 7 rows.

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

Now construct an *infinite* $0, 1$ *string* s^* as follows: for every $k \in \mathbb{N}$, define $s^*[k] := 1 - T[k, k]$. Abusing notation, let s^* also be the real number[1] in $(0, 1]$ given by the map (1). Here is the key fact we will use which follows from the construction of s^*:

$$\text{For any } k \in \mathbb{N},\ s^*[k] \neq T[k, k] \tag{3}$$

And in particular, for any $k \in \mathbb{N}$, the kth bit after the binary point in the binary representation of s^* is **not** $T[k, k]$.

But $s^* \in (0, 1]$; so (2) implies there must exist some $k \in \mathbb{N}$ such that $r_k = s^*$, and in particular, $s^*[k] = T[k, k]$. But this is a contradiction!

Therefore our supposition that the set $(0, 1]$ is countable must be wrong. This establishes the theorem. □

2 Repercussions for Computing

- **Uncomputable Problems.**

  The notion of countable and uncountable sets have deep repercussions in computing. To this end, let is define what a *Boolean function* over natural numbers is.

  Definition 1. A function $f : \mathbb{N} \to \{0, 1\}$ is called a Boolean function. Such a function decides a YES/NO property about every natural numbers. For instance the IsOdd function takes value 1 on the odd numbers and 0 otherwise.

  Let us denote the collection of **all** Boolean functions as $\mathcal{F}$. In your problem set you prove the following theorem.

  Theorem 2. The set $\mathcal{F}$ is not countable.

  Let us denote the set of all Python programs which take input a natural number and outputs 0 or 1 as $\mathcal{P}$. In your problem set you prove that the set $\mathcal{P}$ is countable (in fact, the set of all Python programs, and indeed the set of all strings is countable).

  Theorem 3. The set $\mathcal{P}$ is countable.

  We say a function $f \in \mathcal{F}$ is *computable* if there is some python code $P \in \mathcal{P}$ which has the same I/O behavior as f. That is, for all $n \in \mathbb{N}$, $f(n) = P(n)$. Using the above two theorems, we can conclude the following.

[1] Do you see why this string is *not* the all 0s string? The reason is that the number 1 is actually represented as the all 0's string since it is $1.0000000\ldots$. Since 1 appears as some r_n, we get $T[n, n] = 0$ implying $s[n] = 1$.

Theorem 4. There must exist a Boolean function $f : \mathbb{N} \to \{0,1\}$ which is **not computable**.

Proof. For sake of contradiction, suppose that *all* functions $f \in \mathcal{F}$ are computable. Thus, for every $f \in \mathcal{F}$, we can map it to some $P \in \mathcal{P}$. Note this mapping is injective; we cannot have two different f and g computable by the same python code P. Why? Well if so, then since f and g are different, there is some n such that $f(n) \neq g(n)$; but the code P returns the same answer on n.

Thus, the map $f \mapsto P$ is an *injective map*. Let's call this map $\phi : \mathcal{F} \to \mathcal{P}$.

Now, $\mathcal{P}$ is countable (Theorem 3). Therefore, there is an injection $\psi : \mathcal{P} \to \mathbb{N}$.

Now consider the following function $h : \mathcal{F} \to \mathbb{N}$ defined as $h(f) := \psi(\phi(f))$. That is, we use ϕ to see which python program this function f maps to, and then use ψ to map it to a natural number. This is a *composition* operation. ✍

Exercise: *Prove that $h : \mathcal{F} \to \mathbb{N}$ is injective.*

Since $h : \mathcal{F} \to \mathbb{N}$ is injective, we would conclude that $\mathcal{F}$ is countable as well. But this contradicts Theorem 2. Hence our supposition is wrong. That is, there must exist some function $f \in \mathcal{F}$ which is *uncomputable*. □

- **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 going to be interpreted as an *algorithm* (or a piece of Python code, if you will). The second string is going to be interpreted as an *input* (or data, if you will) to this algorithm.

  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 Python program. 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 5. There is no finite time procedure which for *every* input A and I decides *correctly* whether or not $A(I)$ halts.

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 A which takes input a string B and acts as follows:

L1. First the algorithm run 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 while loop.

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

The above algorithm A is also a string; in particular it contains 3 lines as described above. To be very precise, if we fix Python as our language of choice, then A is the correct Python implementation of the above procedure where $H(\cdot, \cdot)$ has also been implemented in Python. For our contradiction, we ask ourselves what happens when (A, A) is passed to the procedure H.

Case 1: $H(A, A)$ return YES. That is, the procedure H looks at the string A and decides that $A(A)$ *halts*. But now consider the run of $A(A)$ by looking at the code. In Line 1, the code runs $H(A, A)$. Since we have assumed in this case that $H(A, A)$ is YES, the algorithm branches to Line 2. However in Line 2, the run goes into an infinite loop. That is, $A(A)$ *never halts*. In this case, we have reached a contradiction.

Case 2: Perhaps, $H(A, A)$ returns NO. That is, the procedure H looks at the string A and decides that $A(A)$ *never halts*. But now consider the run of $A(A)$ by looking at the code. In Line 1, the code runs $H(A, A)$. Since we have assumed in this case that $H(A, A)$ is NO, the algorithm branches to Line 3. However in Line 3, the run returns 1 and halts. That is, $A(A)$ *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: *As can be imagined, many problems in automated verification is indeed undecidable. That is, no code can exist for checking correctness of some codes. Of course, that doesn't mean computer verification folks packed their bags and left. The above theorem just means there is no procedure which works for everything. It doesn't mean there is no verifier which works* 99.9999999% *of the time.*

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 unknowns, 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.