Logic, Computability and Incompleteness

The Halting Problem and Uncomputability

Idealized Computability

- We have been exploring a highly idealized mathematical characterization of computability in the guise of abstract Turing machines

  in which there is no finite upper bound on input size, tape length, number of instructions defining the program, nor steps in the computation.
- Hence abstract TM's can perform computations that cannot possibly be performed by <u>any</u> physical machine, past, present or future.
- In this respect TM's provide an absolute mathematical limit on what is computable, in principle (assuming C-T Thesis)

Cardinality and Uncomputability

Nonetheless, considerations of brute cardinality reveal that some relevant phenomena must remain outside these highly idealized boundaries, and that even abstract Turing machines cannot compute all functions.

This is because the set of all TMs is enumerable, and hence so is the set of all Turing computable functions,

while, e.g., the set of all functions from positive integers **P** to positive integers is not enumerable.

$$|\{\forall \text{ functions } f : \mathbf{P} \rightarrow \mathbf{P}\}| > |\{\forall f : f \text{ is Turing computable}\}|$$

- So it follows that there must ∃ functions *f*: **P** → **P** which are ***not*** computed by any Turing machine.

Uncomputability via Diagonalization

- Mere cardinality considerations indicate that such functions must exist. We will now define a specific function

  u: $\mathbf{P} \to \mathbf{P}$ such that u cannot be computed by any Turing machine, on pain of contradiction.

  u will be defined using the diagonal method.
- So first need to give an enumeration of all TM's and specify u relative to this list.
- Each TM can be represented as a word, i.e. a finite string of symbols, in the enumerable alphabet

$$\mathrm{R}, \mathrm{L}, S_0, \mathrm{q}_1, S_1, \mathrm{q}_2, S_2, \mathrm{q}_3, S_3, \ldots$$

Enumerate all Turing Machines

- Only a (very!) proper subset of words in this alphabet will represent a TM.
- There are 5 effective constraints for determining which words denote TM's:

  i) <u>length</u> of word is exactly divisible by 4.

  ii) only the symbols q_1, q_2, q_3, … occur in positions

  1, 4, 5, 8, 9, …, 4_n, …, 4_{n+1}, …

  iii) only the symbols S_0, S_1, S_2, … occur in positions

  2, 6, 10, …, 4_{n+2}, …

  iv) only the symbols R, L, S_0, S_1, S_2, … occur in positions

  3, 7, 11, …, 4_{n+3}, …

Enumerate all Turing Machines

v) no configuration of the form q_i, S_k occurs more than once in a word.

- Will now enumerate the set of all words in this alphabet, using only the characters ‘1’ and ‘2’ for coding and then decimal notation to define the ordering on the code:

1	2	3	4	5	6 ….	Enumeration of
R	L	S_0	q_1	S_1	q_2 ….	alphabet.
12	122	1222	12222	122222	1222222 ….	Code for symbol

Enumerate all Turing Machines

- In this manner, the code for the *n*th symbol in the alphabet will be a 1 followed by *n* 2's.
- A word is a concatenation of symbols in the alphabet,

  and the code for a word is the concatenation of the codes for its constituent symbols.
- Finally, the ordering on the words is given by their codes arranged in **decimal notation**.
- This yields a gappy list of words, where w_{12} is the first entry.
- Given the enumeration of words, it is effective (and laborious!) to determine the corresponding list of TM's

Enumerate all Turing Machines

- The first word on the list that specifies a TM (i.e. which satisfies the 5 effective constraints given above) is w_n

  where n = 1222212221212222
- Decoding n we get 4314

  = the string formed by concatenating the 4th 3rd 1st and 4th symbols of the alphabet

  = $q_1\ S_0\ R\ q_1$
- Adopting our standard conventions of interpretation, this TM computes the identity function $i(x) = x$.

Enumerate all Turing Machines

- The second word on the list that specifies a TM is w_m where m = 12222122212212222
- Decoding m we get 4324

  = the string formed by concatenating the 4th 3rd 2nd and 4th symbols of the alphabet

  = $q_1\, S_0\, \mathrm{L}\, q_1$
- Adopting our standard conventions of interpretation, this TM also computes the identity function $i(x) = x$.

Enumerate all Turing Machines

- Can systematically eliminate gaps in the list to get an enumeration of all Turing machines $TM_1, TM_2, TM_3, \ldots.$ which in turn gives an enumeration of all functions of one argument $f_1, f_2, f_3, \ldots.$ such that $f_n : \mathbf{P} \rightarrow \mathbf{P}$.
- Since we now have a list, it's possible to use <u>diagonalization</u> to define the function u on positive integers, such that

$$u(n) = \{1, \quad \text{if } f_n(n) \text{ is undefined}$$
$$= \{f_n(n) + 1 \text{ otherwise}$$

- Now $u : \mathbf{P} \rightarrow \mathbf{P}$ is a perfectly well defined **total** function on positive integers, but it can't appear anywhere on the list as, say, f_m, since this would entail a **contradiction** as follows:

Diagonalize

Suppose u were the mth function f_m on the list, for some arbitrary m. Then by the definition of u,

$$u(m) = \{1, \quad \text{if } f_m(m) \text{ is undefined}$$
$$= \{f_m(m) + 1 \text{ otherwise}$$

But this is impossible, since if $f_m(m)$ is undefined then $u(m) = 1$ and $u(m) \neq f_m(m)$,

and if $f_m(m)$ is defined then $u(m) = f_m(m) + 1$ and again $u(m) \neq f_m(m)$.

So u cannot appear anywhere on the complete list of Turing computable functions, and hence u is *not* Turing computable.

Diagonalize

- Although no TM computes the **total** function u, it's easy to determine its *first few* values.
- As we've already seen, f_1 on the list = f_2 = i = the identity function, so that $f_1(1) = 1$ and $f_2(2) = 2$. Hence $u(1) = 2$ and $u(2) = 3$.
- So why can't we go ahead and effectively determine $u(n)$ for any positive integer n?
- If this were possible, it would refute the Church-Turing thesis by showing that u actually is computable, even though we have already demonstrated that u is not Turing computable.

The Halting Problem

- The essential issue is this: for any given TM on the list, say TM_n, is there an effective procedure for determining whether or not TM_n eventually halts in standard configuration when started in standard initial configuration scanning the leftmost of an unbroken string of *n* 1's?
- In other words, is there an effective procedure for determining whether or not $f_n(n)$ is defined, and if so, what its value is?
- If there is such a procedure, then the function *u* is indeed computable.
- So an equivalent question is, given an arbitrary TM started on an arbitrary input, can we determine in a finite number of steps whether or not it will ever halt?

The Halting Problem

- In simple cases we can often tell just by inspection that a machine will never halt, e.g.

  $q_1\ S_1\ S_1\ q_1$ or $q_1\ S_1\ S_0\ q_1$; $q_1\ S_0\ L\ q_1$
- But what if confronted by, e.g. TM_{15472} on the list, who's program is specified by say, 800 distinct quadruples.
- Is there a computable halting function that can determine the answer?
- It can be demonstrated by direct argument that no such halting function can be computed by any Turing machine.
- Consider the 2-place halting function *h* such that:

The Halting Problem

$h(m,n)$ = {2, if machine number *m* halts on input *n*.

= {1, if it never halts on input *n*.

If *h* were computable then *u* would be computable too.

- Suppose that *h* were computed by some TM, say machine **H**.

Then we could construct the following machine $\mathbf{M}^*$

such that for all positive integers *n*,

$\mathbf{M}^*$ halts on input *n* iff TM_n on the list does not.

In other words, $\mathbf{M}^*$ halts on input *n* iff $h(n,n) = 1$.

$\mathbf{M}^*$ is a composite of 3 machines: a ‘doubler’ which given

$\overset{\wedge}{0}\underbrace{111\ldots1}_{n}0$ as input yields $\overset{\wedge}{0}\underbrace{111\ldots1}_{n}0\underbrace{111\ldots1}_{n}0$ as output

The Halting Problem

- This is then fed into the hypothetical machine **H** as input, and lastly the output from **H** is fed into a ‘halting’ module.
- By the definition of **H**, it will output:

  $0\underline{1}0$ if $h(n,n) = 1$, i.e. TM_n does not halt on input n

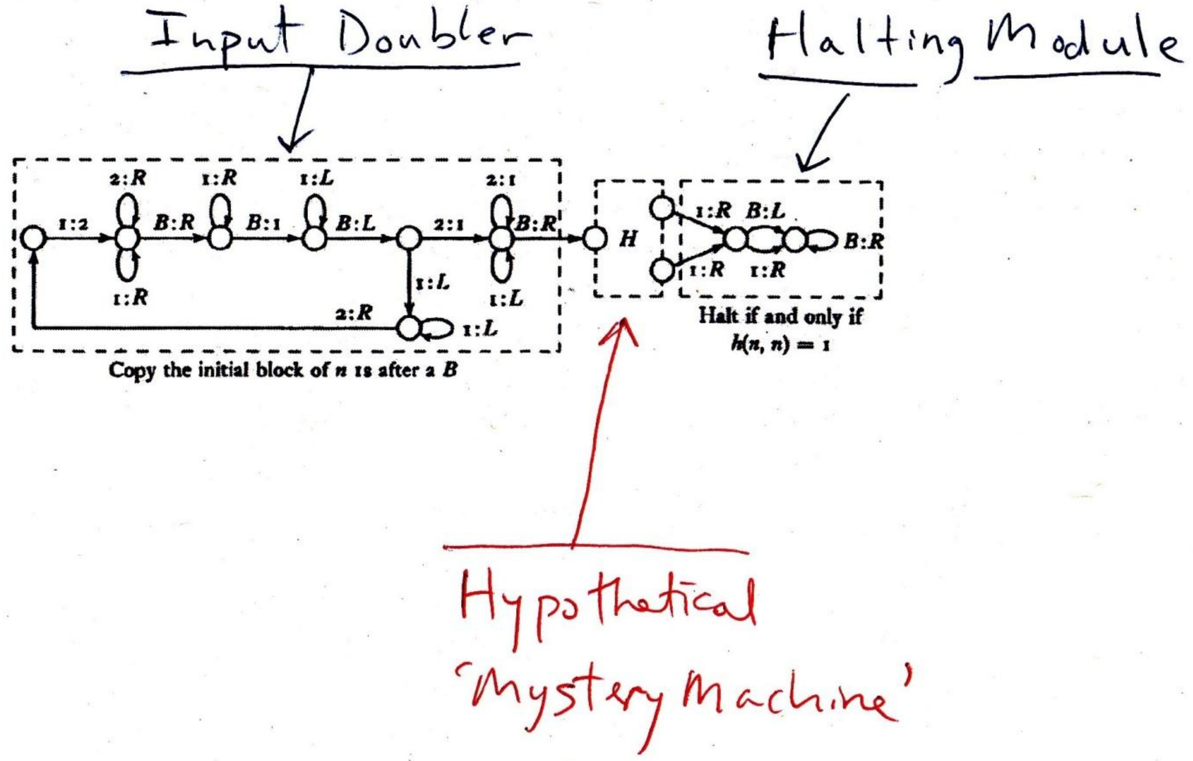
  or

  $0\underline{1}10$ if $h(n,n) = 2$, i.e. TM_n does halts on input n.
- The final ‘halting’ module of $\mathbf{M}^*$ is such that it

  halts on input configuration $0\underline{1}0$ and

  it runs on rightward forever on input configuration $0\underline{1}10$.

Flow Graph of M*



Copy the initial block of n 1s after a B

Halt if and only if $h(n, n) = 1$

Self Destruct!

- Thus for all positive integers n, **M*** halts on input n

  if and only if the nth TM on the list does not halt on n.

  But, since **M*** is a TM, it is the kth machine on the list for some positive integer k. So it is $\mathbf{M}^*_k$ in the ordering.

  By construction, $\mathbf{M}^*_k$ must halt on input k

  if and only if it does not halt on input k!

- Hence no such machine **M*** can exist.

  But the 'doubler' and 'halting' modules obviously do exist.

  Conclusion: the hypothetical machine **H** cannot exist

  and so the halting function h is not Turing Computable.

Back to the Beginning

- And if the Church-Turing Thesis is true, then the halting problem is absolutely unsolvable.
- An enumeration of a set $\mathbf{\Gamma}$ is effective iff $\mathbf{\Gamma}$ is finite, or else there is an explicit, mechanical procedure for determining the value $f(n) \in \mathbf{\Gamma}$ in a finite number of steps, for every $n \in \mathbf{P}$.
- In the first lecture it was promised that we would encounter sets that are enumerable (because of their denumerable cardinality), but not effectively enumerable.
- Here's our first example: the set of Turing computable functions on the positive integers which are not total functions.