Logic, Computability and Incompleteness

Turing Machines and Computability

What is Computation?

- Two basic ways to look at **computation**:

  1) In practice, as the activity of *physical machines*, as what computers **do**, where a *computer* is an actual device that exists in space and time

  2) In theory, as a certain type of *mathematical 'process'* – as something that belongs to an abstract, idealized domain, like Euclidean geometry, set theory or algebra.

  Our amazing present day *physical* computing machinery was only made possible by prior conceptual advances in the abstract *mathematical theory* of computation (MTC)

- But before an advanced MTC was developed (19th-20th century) first there were relatively simple calculational devices

Early Practice

- ‘Calculate’ comes from the Latin word for ‘stone’, and in ancient times, simple calculations were performed using piles of pebbles or *calculi* to **represent numbers**
- Abacus: ingenious frameset with movable beads on rods to make rapid calculations
- From the 17th century onwards there were assorted geared mechanisms for making calculations:
  systems of cogs and levers that could add, multiply, etc.
- As well as wheel and disk ‘differential analysers’
- This style of approach culminated in the **Analytical Engine** of Charles Babbage in the mid 1800’s – the most powerful and innovative computing machine envisioned up to that time.

Early Theory

- The foregoing examples are purely applications-driven, and there was no overarching theory of computation
- The German philosopher and mathematician Gottfried Leibniz (1646-1716) entertained the idea of a **Universal Logical Language** based on an '*alphabet of human thought*', which could be used to derive all possible truths through basic combinatorial operations, a '*calculus ratiocinator*'.
- Leibniz's idea, though visionary (and highly optimistic!), remained more or less a futuristic dream.
- Then in the late 19th century, another German mathematician and philosopher, Gottlob Frege (1848-1925) developed a system of **formal logic** in an attempt to provided a rigorous

foundation for mathematics

Frege Invents Formal Syntax

- Frege's *Begriffsschrift* (1879) introduced Universal and Existential quantification, along with the rest of the technical machinery now known as First-Order Logic.
- The logical system of the *Begriffsschrift* is the first instance of an **artificial language** constructed according to exact rules of syntax.
- This logical calculus allowed *formal proofs* to proceed as '*computations*' in accordance with a fixed set of inference rules,

  as a serious of purely syntactic manipulations

  which in principle encompassed all the reasoning ordinarily used in mathematics.

Effective Procedures

- And because the system is purely **formal**, the rules can be applied without any specification of their intended **meaning**.
- This idea is **fundamental**, because central to the **mathematical theory of computation** is the intuitive notion of an effective (or ‘mechanical’) procedure, or algorithm.

which is simply a **finite set of instructions** for syntactic manipulations,

that can be followed by a human being who is capable of carrying out only very elementary operations on symbols

or by potentially by a ***machine***…

Effective Procedures

- A key constraint is that the machine or the human can follow the rules without knowing what the symbols *mean*.
- The notion of an effective procedure is very general –

  it doesn't specify what form the instructions should take,
  what the manipulated symbols should look like,
  nor precisely what manipulations are involved.
- The underlying restriction is simply that the set of rules is finitary and can proceed formally,

  i.e. without any additional interpretation or understanding.

Turing Machines

- So there are any number of different possible frameworks for filling in the details and making the broad intuitive idea precise.
- The British mathematician Alan Turing (1912-1954) introduced one such framework in his groundbreaking 1936 paper "On Computable Numbers"
- His "automatic computing machines", now generally referred to as "Turing Machines", supply a very intuitive and elegant rendition of the notion of an effective procedure
- But there is a variety of well known alternative frameworks, including Church's Lambda Calculus, Gödel's Recursive Function Theory, Lambek's Infinite Abacus Machines, etc.

Turing Machine Basics

- In turn there are different ways of specifying TMs – we'll use the conventions adopted in B&J. The first thing we need is
- (1) a **tape**: divided into squares or cells, unbounded in both directions.

  All but a finite number of cells of the tape are blank.
- (2) a **finite list of symbols** S_1, S_2, …, S_n. Non-blank cells are filled with exactly one symbol from the list.
- We treat 'blank' as the symbol S_0.
- (3) a **read/write head**: which can scan exactly one cell of the tape at a time.

Turing Machine Basics

- (4) a **finite set of states**: which are conditional instructions saying *what to do* for a given scanned input symbol.
- What can a TM do? There are n+4 possible *overt* actions:
- (i) **Nothing** (= halt)
- (ii) **Move L** one square
- (iii) **Move R** one square
- (iv) **Write** S_0 in place of currently scanned symbol
- (v) **Write** S_1 in place of currently scanned symbol

⋮

- (n+4) **Write** S_n in place of currently scanned symbol

Turing Machine Basics

- Plus one *covert* action: **enter next state**.
- The **present state** plus **scanned symbol** functionally determine the next *overt* and *covert* action.
- There are various ways of representing the finite set of states which constitute the program of instructions, including:
- Machine Table, Flow Diagram, (finite) Set of Quadruples
- We'll focus on Set of Quadruples scheme of representation, because it allows TM's to be easily enumerated.
- E.g. q_1 S_0 R q_2 where q_1 is the current state, S_0 the scanned symbol, R is the overt act of moving right one square and q_2 the covert 'act' of entering state q_2.

TM Examples

- So the quadruple q_1 S_0 R q_2 is a conditional instruction:

  if in state q_1 reading the symbol S_0, **then** move R one square and enter state q_2
- If a TM is in current state q_i scanning a symbol S_k and there is no quadruple in its program of instructions beginning with the pair q_i S_k then the machine can <u>do</u> <u>nothing</u> and must **halt**.
- Another example: q_2 S_1 S_0 q_1 where q_2 is the current state, S_1 the scanned symbol, S_0 is the overt act of printing a blank (i.e. erasing S_1) and q_1 the covert act of entering state q_1.
- Note that the next state entered doesn't have to be new, so that

  q_2 S_1 L q_2 is a perfectly good quadruple.

TM Examples

- Note also that any (non-empty) finite set of perfectly good quadruples constitutes a perfectly good Turing Machine.
- So each of the single quadruples above ***is*** a particular TM.
- Here's another example of a single instruction TM:

$$q_1\ S_1\ S_1\ q_1$$

- Any particular **computation** on a given input string can be described as a sequence of configurations,

where a configuration shows the **content of the tape** at that stage in the sequence,

and it shows what **state the machine is in**

and which **square is being scanned**.

TM Examples

- A handy convention: write 'B' for the symbol S_0, and write '1' for the symbol S_1
- Here's an example of a TM that, if started in state q_1 scanning the leftmost of an unbroken string of 1's on an otherwise blank tape, will halt scanning a 0 if the number of 1's in the string is <u>even</u>, and will halt scanning a 1 if the number of 1's is <u>odd</u>:

q_1 1 B q_2

q_2 B R q_3

q_3 B 1 q_5

q_3 1 B q_4

q_4 B R q_1

TM Examples

So this machine gives a yes/no answer to the question
‘Is the number of 1’s in the input string odd?’

- Following are 2 sample **computations**
  (= sequence of configurations) in parallel:

$q_1 1 B q_2$; $q_2 B R q_3$; $q_3 B 1 q_5$; $q_3 1 B q_4$; $q_4 B R q_1$

Start	$\ldots 00\underline{1}100 \ldots$	$\ldots 0\underline{1}110 \ldots$	
	1	1	
	$00\underline{0}10$	$0\underline{0}110$	
	2	2	
	$000\underline{1}0$	$00\underline{1}10$	
	3	3	
	$000\underline{0}0$	$00\underline{0}10$	
	4	4	
	$0000\underline{0}$	$000\underline{1}0$	
	1	1	
	Halt	$000\underline{0}0$	
		2	
		$0000\underline{0}$	
		3	
		$0000\underline{1}$	
		5	Halt

Abstract versus Physical

- TM’s are mathematical abstractions, not actual machines. They don’t exist in physical space-time, and they have no causal powers.
- Hence TM computations have no chronological duration or physical extension.
- Indeed, our picturesque images of tape, moving read/write head, etc. are just conceptual aids and not essential to a TM’s mathematical structure.
- In order to perform *actual* computations, an abstract Turing machine must be realized or implemented by a suitable arrangement of mass/energy.

Multiple Realization

- And as Turing observed long ago, there is no privileged or unique way to do this.
- Like other abstract structures, such as chess games and isosceles triangles, Turing machines are *multiply realizable*
- What unites different types of physical implementation of the same abstract TM is nothing that they have in common as physical systems, but rather a structural isomorphism characterized at a higher level of description.
- Hence it's possible to implement the very same computational formalism using modern electronic circuitry,

  a human being executing the instructions by hand with paper and pencil,

Multiple Realization

a Victorian system of gears and levers,

as well as more atypical arrangements of matter and energy including rolls of toilet paper serving as tape and beer cans for the symbol ‘1’.

- In any case, only particular physical realizations can perform actual computations, and any such realization must perforce be just a limited **approximation** of the idealized abstract TM.
- This is because physical devices will always have finite bounds, while abstract TM’s **do not**.
- There is no finite upper bound on input size, tape length, number of instructions defining the program, nor steps in the computation.

Syntax versus Semantics

- As in our original characterization of an effective procedure, the program of instructions specifying a TM is merely a recipe for syntax manipulation, since

  the essence of classical computation **is rule governed symbol manipulation**.
- Additionally, TM's can be interpreted as computing various numerical functions by (i) interpreting the input/output strings as *notation for numbers*, and (ii) supplying conventions governing *permissible configurations*.
- We will be especially interested in interpreting TM's as computing functions from positive integers to positive integers.

Some relevant conventions:

1) positive integers are represented in monadic notation.

2) TM's read and write only the symbols B and 1

3) Permissible initial configuration: starts in lowest numbered state reading leftmost 1 on the tape.

Input numbers are separated by single blanks.

4) Permissible halting configuration: halts reading the leftmost of an unbroken string of 1's. This gives the output value.

If it doesn't halt, or halts in a non-permissible configuration, then the computed function is <u>undefined</u> for that input.

- In this manner we can specify the class of

Turing Computable Functions on the positive integers.

- For example, here’s a TM that computes **addition** under the foregoing conventions:

  $q_1 1 B q_1$

  $q_1 B R q_2$

  $q_2 1 R q_2$

  $q_2 B 1 q_3$

  $q_3 1 L q_3$

  $q_3 B R q_4$

Machine $q_1 1 B q_1$; $q_1 B R q_2$; $q_2 1 R q_2$; $q_2 B 1 q_3$; $q_3 1 L q_3$; $q_3 B R q_4$ computes 2+3 = 5

Start …0$\underline{1}$101110…

1

…0$\underline{0}$101110…

1

…00$\underline{1}$01110…

2

…001$\underline{0}$1110…

2

…001$\underline{1}$1110…

3

…00$\underline{1}$11110…

3

…0$\underline{0}$111110…

3

…00$\underline{1}$11110…

4 Halt

Variations on a Theme

- We have looked at **one** common way of formulating TM's, but there are a number of possible variations, all of which are provably equivalent:

  1) Allow tape to be 'infinite' in **one** direction only. This might appear to restrict computational capacity, but it doesn't.

  2) Instead of a single tape, have **multiple tapes and heads**.

  3) Use **two-dimensional tape**, 'infinite' along both dimensions.

  4) Allow read/write head to **move some** (fixed) **arbitrary number of squares** at each transition, not just one

  still no increase in power.

Variations on a Theme

5) Instead of arbitrarily large finite alphabet S_1, S_2, …, S_n, **restrict to only two symbols**, ‘0’ and ‘1’ (as in conventions above).

6) Turing originally used a **quintuple formulation** rather than a set of quadruples, wherein the machine can both *write and move its head* in the same transition. This doesn’t increase power.

7) **Non-deterministic TM’s**: Allow distinct quadruple instructions for the same *current state/scanned symbol* pairs.

This leads to multiple possible transitions from the same configuration.

Universal Turing Machines

- In some sense, a standard TM can be thought of as *dedicated* to computing a particular function. This is a limitation
- One of Turing's great insights concerns the existence of *Universal Turing Machines* (**UTM**).
- When started on a tape containing the encoding of another Turing machine, call it *T*, followed by the input to *T*, a **UTM** produces the same result as *T* would when started on that input.
- Essentially a **UTM** can simulate the behavior of **any** Turing machine (including itself).
- So **UTM** is the conceptual prototype for a **programmable computer**, where the inputted encoding of the machine to be imitated serves as the program.

Church's Thesis and Computability

- The Church-Turing Thesis (or Church's Thesis):

  Any function which is in principle computable can be computed by some Turing Machine.
- So if Church's Thesis is **true**, then the Turing computable functions are co-extensive with the class of functions which can be computed by **any** effective procedure whatever.
- Hence according to the thesis, TM's capture a maximal, stable class of phenomena, a basic 'mathematical kind'.
- Thus, our notion of 'in principle **computability**' can be nicely expressed as '**computable by a UTM with no finite upper bound on length of program, tape or running time**.'

Church’s Thesis

- Church’s Thesis cannot be proved, because there is no limit on the number of different frameworks in which the intuitive notion of an effective procedure might be formally expressed.
- Future renditions of the notion cannot currently be foreseen…
- However, Church’s Thesis is **refutable** *if false*.
- So we can gather ‘inductive evidence’ for the truth of the thesis by comparing different frameworks.
- Thus far, every given framework for characterizing the notion of an effective procedure has been shown to be equivalent to Turing computability.

Alternative Frameworks

- So in support of Church's Thesis, and to explore some (seemingly) very different methods, we'll investigate two other computational frameworks.
- First we'll look (briefly) at Abacus computable functions, which can be shown to be Turing computable, i.e. **A** ⊆ **T**.
- Then we'll look at Recursive functions, which can be shown to be Abacus computable, i.e. **R** ⊆ **A**.
- Finally, it can be shown that **T** ⊆ **R**, thereby closing the chain of inclusions **R** ⊆ **A** ⊆ **T** ⊆ **R**, to establish that **R** = **A** = **T**.

Abacus Machines

- In contrast to TM's, Abacus Machines bear a much closer resemblance to (seemingly more powerful) high speed digital computers, because AM's have an unlimited amount of **random access storage**.
- AM's have an **unbounded number** of registers R_0, R_1, R_2,…, in each of which can be written numbers of **arbitrary size**.
- An AM has 'random' access to these registers, in the sense that it can go directly to register R_n, without inching ('blindly') along a tape square by square.
- Each register R_n has its own address n, which allows the machine to carry out operations such as 'compute the sum of the numbers in registers R_n and R_m and store it in register R_k'

Abacus Machines

- Our pictorial representation of such a machine will depict the registers as **labelled boxes** which can containing arbitrarily large quantities of **stones**.
- The **boxes** are located in a large cave, and on the floor of the cave is an arbitrarily large supply of **stones**.
- There is a caveman executing the program, and the possible actions the caveman can perform are:

  i) pick up *one stone* and **add** to box *n*

$$\begin{tikzcd} {} \arrow[d] \\ (n+) \arrow[d] \\ {} \end{tikzcd}$$

Abacus Machines

ii) **remove** *one stone* from box *n*, **if** it is not empty.

↓

(*n*-)

/ \ e

- So program the caveman follows is given by a flow diagram with arrows connecting these actions on registers,

  where the minus nodes are branch points and can have two exit arrows,

  and so the next action in the sequence will depend on whether or not the box is already empty.

Abacus Machines

- In this manner, can perform basic operations such as the above 'compute the sum of the numbers in registers R_n and R_m and store it in register R_k'

$$[m] + [n] \longrightarrow k$$

where '$[m]$' denotes the number of stones in box m.

- With these resources can construct AM's to compute multiplication on the natural numbers, exponentiation, etc.
- See B&J chapter 6 for relevant technical details.

Abacus Computable Functions are Turing Computable

- In order to show that **A** ⊆ **T**, need to give a method which, applied to any AM flow graph yields the flow graph of a TM that computes the same function.
- **Basic plot**:

  1) give method for correlating AM registers with contents of TM tape.

  2) replace each (*s*+) node with a TM flow graph which finds the appropriate block of 1's on the tape, adds a 1 to the R, shifts all remaining blocks to R, then returns to L-most 1 on tape.

Abacus Computable Functions are Turing Computable

3) replace (*s*-) nodes with corresponding flow graph:

finds R-most 1 in *s*th block, erases, shifts remaining blocks, returns to L-most 1 on tape.

4) connect each remaining loose arrow to a ‘mop-up’ graph which keeps L-most 1 on tape, erases everything else on tape except the *n*th block (which contains output value), shifts this block to L-most position and halts in standard configuration

(again, see B&J chapter 6 for relevant technical details).

This will then suffice to show that **A** ⊆ **T**.

- **Moral of the story**: random access storage does not increase in principle computing power, but only speed.