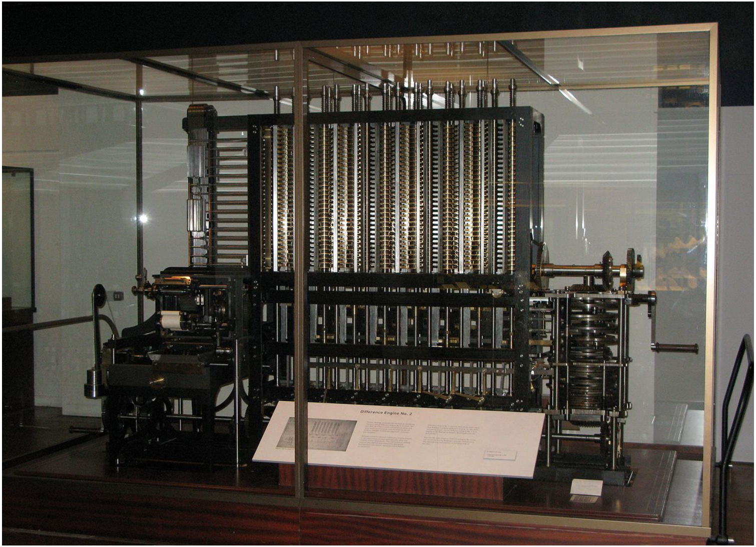
ITCS 532:
3. Universal Turing Machines

Rob Egrot

Universal Computation

- Machines for computation have existed for thousands of years. E.g.
  - The Antikythera mechanism from 1st or 2nd century BC Greece.
  - The Banu Musa brothers' automatic mechanical flute player from 9th century Persia.
- These devices can compute, but they cannot *simulate*.
- I.e. a modern computer can mimic the calculations of the Antikythera mechanism, but the Antikythera mechanism can't run Windows.
- The first *universal* computer design was probably the Analytical Engine by Charles Babbage and Ada Lovelace in 1837.
- This was never built due to cost, but it would, in theory, have been able to run Windows.

Difference Engine no. 2



The Difference Engine no. 2. Photo from Wikipedia (https://commons.wikimedia.org/w/index.php?curid=4807331)

Universal Computation with Turing Machines

- Babbage and Lovelace did not have the theoretical framework to understand universal computation.
- But we do, because we know about Turing machines.
- Remember that a Turing machine is defined by a 5-tuple $(Q, \Sigma, q_0, H, \delta)$.
- We've been defining alphabets and states on an ad hoc basis, but we can make our approach more systematic by unifying our choice of symbols.
- Define $\hat{\Sigma} = \{\sigma_0, \sigma_1, \sigma_2, \ldots\}$ and $\hat{Q} = \{q_0, q_1, q_2, \ldots\}$.
- This is enough to define any TM.

Encoding Turing Machines

- Given a TM defined by $(Q, \Sigma, q_0, H, \delta)$ we can assume:
- Q is a finite subset of $\hat{Q}$, Σ is a finite subset of $\hat{\Sigma}$.
- q_0 is just q_0 from $\hat{Q}$.
- $:$, $\sqcup$, $\leftarrow$, and $\rightarrow$ are σ_0, σ_1, σ_2, and σ_3 from $\hat{\Sigma}$ respectively.
- Every Turing machine is specified by a finite subset of $\hat{Q} \cup \hat{\Sigma}$, along with a transition function δ that is formally a finite subset of $\hat{Q} \times \hat{\Sigma} \times \hat{Q} \times \hat{\Sigma}$.
- So the set of all Turing machines corresponds to a subset of the set of all finite subsets of a countable set, and so is countable.
- I.e. there's a 1-1 function **code** from the set of all Turing machines to $\{0, 1\}^*$.

Defining a **code** function - symbols and states

- There are many ways we can define a **code** function.
- Here is one example.
- For q_n we define $\textbf{code}(q_n)$ to be $n+1$ '1' symbols. So, e.g. $\textbf{code}(q_2) = 111$.
- For σ_n we define $\textbf{code}(\sigma_n)$ to be $n+1$ '1' symbols. So, e.g. $\textbf{code}(\sigma_1) = 11$.
- Given a Turing machine T with states $Q = \{q_0, \ldots, q_m\}$ and alphabet $\Sigma = \{\sigma_0, \ldots, \sigma_n\}$ we define the strings

$$\textbf{code}(Q) = \textbf{code}(q_0)0\textbf{code}(q_1)0\ldots0\textbf{code}(q_m)0$$

and

$$\textbf{code}(\Sigma) = \textbf{code}(\sigma_0)0\textbf{code}(\sigma_1)0\ldots0\textbf{code}(\sigma_n)0$$

Defining a **code** function - accept and reject

- Let q_i and q_j be the accept and reject states respectively.
- The concatenated string

$$\mathbf{code}(Q)0\mathbf{code}(\Sigma)0\mathbf{code}(q_i)0\mathbf{code}(q_j)0$$

  encodes the states and alphabet that define T.
- All that is left is to find a way to encode δ.

Defining a **code** function - δ

- δ is formally defined as a set of tuples (q, σ, q', σ'). We can code a tuple $t = (q, \sigma, q', \sigma')$ as

$$\mathbf{code}(t) = \mathbf{code}(q)0\mathbf{code}(\sigma)0\mathbf{code}(q')0\mathbf{code}(\sigma')0$$

- If δ is defined by tuples $t_1, t_2, \ldots, t_k$ we can define

$$\mathbf{code}(\delta) = \mathbf{code}(t_1)\mathbf{code}(t_2)\ldots\mathbf{code}(t_k)$$

- Note that this is not strictly speaking well defined, because it depends on the order of the tuples δ.
- To avoid this problem we assume a canonical ordering (something like alphabetical).

Defining a **code** function - putting it all together

- Combining all this we can encode T with

$$\mathbf{code}(T) = \mathbf{code}(Q)0\mathbf{code}(\Sigma)0\mathbf{code}(q_i)0\mathbf{code}(q_j)0\mathbf{code}(\delta)$$

- If I is a string defined by $I = \sigma'_1\sigma'_2\ldots\sigma'_l$ (where the σ' are symbols from $\hat{\Sigma}$) we can encode I using

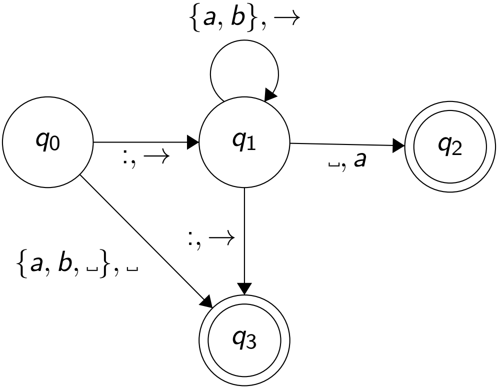
$$\mathbf{code}(I) = \mathbf{code}(\sigma'_1)0\mathbf{code}(\sigma'_2)0\ldots0\mathbf{code}(\sigma'_l)$$

- We can encode the pair (T, I) representing the Turing machine T and input I using

$$\mathbf{code}(T, I) = \mathbf{code}(T)00\mathbf{code}(I)$$

Example

Alphabet $\Sigma = \{:, \sqcup, a, b\}$, machine adds a to the end of its input.



The accept state is q_2, and the reject state is q_3. To encode this machine we first assign a and b correspondents in $\hat{\Sigma}$. We'll say $a = \sigma_4$ and $b = \sigma_5$ (because $: = \sigma_0$, $\sqcup = \sigma_1$, $\leftarrow = \sigma_2$, and $\rightarrow = \sigma_3$).

Example - the formal transition function

The transition function δ is then defined by the tuples

$$
\begin{aligned}
(q_0, :, q_1, \rightarrow) &= (q_0, \sigma_0, q_1, \sigma_3) \\
(q_0, \text{␣}, q_3, \text{␣}) &= (q_0, \sigma_1, q_3, \sigma_1) \\
(q_0, a, q_3, \text{␣}) &= (q_0, \sigma_4, q_3, \sigma_1) \\
(q_0, b, q_3, \text{␣}) &= (q_0, \sigma_5, q_3, \sigma_1) \\
(q_1, :, q_3, \rightarrow) &= (q_1, \sigma_0, q_3, \sigma_1) \\
(q_1, a, q_1, \rightarrow) &= (q_1, \sigma_4, q_1, \sigma_3) \\
(q_1, b, q_1, \rightarrow) &= (q_1, \sigma_5, q_1, \sigma_3) \\
(q_1, \text{␣}, q_2, a) &= (q_1, \sigma_1, q_2, \sigma_4)
\end{aligned}
$$

Note that I didn't put these tuples into alphabetical order first. Technically this is wrong, but we won't worry about that here!

Example - the coded form

So using our definitions we get

- $\textbf{code}(Q) = 10110111011110$
- $\textbf{code}(\Sigma) = 101101110111101111101111110$
- $\textbf{code}(q_i) = 111$
- $\textbf{code}(q_j) = 1111$
- Finally

$$\begin{aligned}\textbf{code}(\delta) =\, &1010110111101011011110110101111101111011010\\ &111110111101101101011110110110111110110111\\ &10110111111011011110110110111011111 0\end{aligned}$$

- And so

$$\begin{aligned}\textbf{code}(T) =\, &1011011101111001011011101111011111011111100\\ &1110111101010110111101011011110110101111101\\ &1110110101111110111101101101011110110110111\\ &11011011110110111111011011110110110111011111 0\end{aligned}$$

Universal Turing Machines

- We can code Turing machines and inputs as finite strings over the alphabet $\{0, 1\}$.
- Turing machines can manipulate finite strings.
- Therefore we can create Turing machines that act on the codes of other Turing machines.
- We're interested in TMs that *simulate* the action of another TM on an input.
- A Turing machine that can do this is called *universal*.
- More precisely, if U is a universal Turing machine (UTM), T is any other TM, and I is an input for T, then:
  - $U(\mathbf{code}(T, I))$ halts if and only if $T(I)$ halts (and accepts or rejects appropriately).
  - The output of $U(\mathbf{code}(T, I))$ is the coded form of the output of $T(I)$.

Designing a Universal Turing Machine

- We will use a 3-tape machine.
- We know that if this exists there's a 1-tape machine that does the same thing.
- The 1st tape is used for input and output. It starts with $\mathbf{code}(T, I)$ written on it (where T is the machine to be simulated on input I). Later in the calculation it will store the state of the tape of T in coded form.
- The 2nd tape will be used to store $\mathbf{code}(T)$ for reference.
- The 3rd tape will be used to store the coded form of the state of T.

Running our UTM - starting up

A run of this machine U on input $\textbf{code}(T, I)$ starts as follows:

1. U copies $\textbf{code}(T)$ from tape 1 onto tape 2.
2. U erases $\textbf{code}(T)$ from tape 1 and shifts $\textbf{code}(I)$ to the start of the tape so tape 1 just contains $\textbf{code}(I)$.
3. U writes $\textbf{code}(q_0)$ on tape 3.

Running our UTM - simulating a step

The simulation of a single step of $T(I)$ by U goes as follows:

1. U searches tape 2 for an instruction corresponding to the state of the machine coded on tape 3 and the symbol currently being read in coded form on tape 1.
   - E.g. symbol being read by T corresponds to string of ones to the left of zero being read by U.
2. Based on the instruction it updates tape 3 to represent the new state of T, and updates tape 1 to represent the new contents of the tape of T and the new position of the tape head.
3. If T has reached a halt state then U halts in the corresponding halt state, otherwise U goes back to 1.

The Existence of Undecidable problems

- Every decision problem corresponds to a formal language.
- Given a finite alphabet Σ the set Σ^*, is countably infinite.
- The formal languages over Σ are the subsets of Σ^*, so the set of all formal languages over Σ is $\wp(\Sigma^*)$, which is uncountable.
- Every Turing machine can be represented by a finite string over the alphabet $\{0, 1\}$.
- So the set of all distinct Turing machines is a subset of the set of all finite strings over $\{0, 1\}$, i.e. $\{0, 1\}^*$, which is countably infinite.
- So there must be (many) more formal languages (and so decision problems) than there are Turing machines capable of deciding them!
- There must be decision problems which cannot be decided (or semidecided) by a Turing machine.

Universal Computers in the Wild

- It turns out that abstract universal computers are actually very common.
- Many if not most deterministic processes that are not obviously trivial turn out to be capable of universal computation.
  - In the sense that you can decide rules for input and output that let you simulate Turing machines (including UTMS).
- We will look at some examples now.

Rule 110

- An elementary cellular automaton consists of a 2-way infinite tape whose cells can contain either 0 or 1.
- At each step of a computation the contents of a cell changes based on its contents and the contents of its immediate neighbours.
- Rule 110 is an elementary cellular automaton with update rules described in the following table.

Cell configuration	111	110	101	100	011	010	001	000
New contents of center cell	0	1	1	0	1	1	1	0

- It can be shown that Rule 110 is capable of universal computation.
- Have to define input and output.
- Note that Rule 110 can't halt, so we have to adjust the definition of 'computation' a bit to take this into account.

Conway's Game of Life

- The Game of Life, invented by John Conway, is another example of a cellular automaton.
- Not elementary as it has an infinite grid.
- Again each cell contains either 0 or 1. If a cell contains 1 then it is 'live' and if it contains 0 then it is 'dead'.
- The update rules are as follows:
  1. A live cell with fewer than two live neighbours dies.
  2. A live cell with two or three live neighbours stays live.
  3. A live cell with four or more live neighbours dies.
  4. A dead cell with exactly three live neighbours becomes live.
- Also capable of universal computation.
- Industrious players have created systems for producing many kinds of behaviour within the Game.
- You can play around with it at `https://bitstorm.org/gameoflife/`.

Langton's Ant

- Langton's ant is essentially a Turing machine variant with an infinite 2D grid instead of a tape.
- Every square of the grid can be black or white.
- The 'ant' is the tape head.
- The ant can face in four directions, up, down, left, and right.
- The movement of the ant is very simple.
  - If the ant is in a white square it turns to the right, colours its square black, then moves forward one square.
  - If the ant is in a black square it turns left, colours its square white, then moves forward one square.
- Langton's ant is a universal computer.
- All known starting configurations lead to eventually creating an orderly 'highway'.
- True for all starting configurations?
- `https://sciencedemos.org.uk/langton_ant.php`