COMP30026 Models of Computation

Decibale and Undecibale Problems

Bach Le / Anna Kalenkova

Lecture Week 11. Part 2

Semester 2, 2021

Decidable Problems

Alan Turing

Alan Turing was born in 1912. At that time, “computer” was a job title: a human employed to do tedious numerical calculations.

Legacy: “Turing machine”, the “Church-Turing thesis”, “Turing reduction”, the “Turing test”, the “Turing award”, and much more.

One of Turing’s great accomplishments was to put “computability” on a firm foundation and to establish that certain important problems do not have an algorithmic solution.

We Have Many Models of Computability

Turing machines (A. Turing, 1936)

Lambda calculus (A. Church, 1936)

Partial recursive functions (S. Kleene, 1936)

Post systems (E. Post, 1943)

Markov algorithms (A. Markov, 1954)

While programs

Register machines

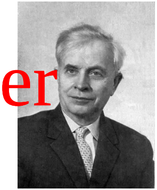
Horn clauses

⋮

Church

Kleene

Post



Markov

The Church-Turing Thesis

The class of computable functions is exactly the class of functions that can be realised by

⟨insert your favourite model here⟩

External evidence: All the above models are "equivalent" in spite of the fact that they all look very different, and were developed independently.

Internal evidence: It seems that no matter how we "extend" any of them, we fail to get something that is more powerful.

Decidable Problems

We can phrase these problems as language decidability problems.

For example, the acceptance problem for DFAs is whether, given a DFA D and a string w, D accepts input w.

Since we can encode the DFA as a string, the acceptance problem can be seen as testing for membership of the language

$$A_{DFA} = \{\langle D, w\rangle \mid D \text{ is a DFA that accepts } w\}$$

By $\langle D, w\rangle$ we mean a (string) encoding of the pair D, w.

DFA Acceptance Is Decidable

Theorem: A_{DFA} is a decidable language.

Proof sketch: The crucial point is that it is possible for a Turing machine M to simulate a DFA D.

M finds on its tape, say

$$\underbrace{1\ldots n}_{Q}\#\#\underbrace{\mathtt{ab}\ldots\mathtt{z}}_{\Sigma}\#\#\underbrace{1\mathtt{a}2\#\ldots\#n\mathtt{b}n}_{\delta}\#\#\underbrace{1}_{q_0}\#\#\underbrace{3\;7}_{F}\#\#\underbrace{\mathtt{baa}\ldots}_{w}\$$$

First M checks that the first five components represent a valid DFA, and if not, rejects.

Then M simulates the moves of D, keeping track of D's state and the current position in w, by writing these details on its tape, after \$.

When the last symbol in w has been processed, M accepts if D is in a state in F, and rejects otherwise.

TMs as Interpreters

We won’t give the details of how the Turing machine simulates the DFA. Many tedious low-level programming steps are involved.

However, it should be clear that it is possible for a Turing machine to mimic DFA behaviour this way.

The description of D is nothing but a “program” and the claim is that a Turing machine can act as an interpreter for this language.

Turing machines themselves can be encoded as strings, and then a Turing machine can interpret Turing machines.

This is no more strange than the fact that we can write an interpreter for Haskell, say, in Haskell.

NFA Acceptance Is Decidable

Theorem:

$$A_{NFA} = \{\langle N, w\rangle \mid N \text{ is an NFA that accepts } w\}$$

is a decidable language.

Proof sketch: The procedure we gave for translating an NFA to an equivalent DFA was mechanistic and terminating, so a halting Turing machine can do that job.

Having written the encoding of the DFA on its tape, the Turing machine can then "run" the machine M from the previous proof.

DFA Equivalence Is Decidable

Theorem:

$$EQ_{DFA} = \{\langle A, B\rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$$

is decidable.

Proof sketch: We previously saw how it is possible to construct, from DFAs A and B, DFAs for $A \cap B$, $A \cup B$, and A^c.

These procedures are mechanistic and finite—a halting Turing machine M can perform them.

Hence from A and B, M can produce a DFA C to recognise

$$L(C) = \left(L(A) \cap L(B)^c\right) \cup \left(L(A)^c \cap L(B)\right)$$

Note that $L(C) = \emptyset$ iff $L(A) = L(B)$.

So M just needs to use the emptiness checker on C.

Generation by CFGs Is Decidable

Theorem:

$$A_{CFG} = \{\langle G, w\rangle \mid G \text{ is a CFG that generates } w\}$$

is decidable.

The proof relies on the fact that we can rewrite any CFG to a particular equivalent form, Chomsky Normal Form.

In Chomsky Normal Form, each production takes one of two forms:

$$A \to B\ C \qquad \text{or} \qquad A \to a$$

(With one exception:
We also allow $S \to \epsilon$, where S is the grammar's start variable.)

Generation by CFGs Is Decidable

For every grammar in Chomsky Normal Form form, if string w can be derived then its derivation has exactly $2|w| - 1$ steps.

So to decide A_{CFG}, we can simply try out all possible derivations of that length, in finite time, and see if one generates w.

Every CFL Is Decidable

Two slides back we saw that it is decidable whether a CFG G generates a string w.

The decider, call it S, took $\langle G, w \rangle$ as input.

Now we are saying that any particular CFL L_0 is decidable:

Theorem: Every context-free language L_0 is decidable.

Proof: This is just saying that we can specialise the decider S. Let G_0 be a CFG for L_0. The decider for L_0 simply takes input w and runs S on $\langle G_0, w \rangle$.

Every CSL Is Decidable

Theorem: For every context-sensitive language L there is a linear bounded automaton (TM with a bounded tape) M, such that M recognises L.

Theorem: If M is a linear bounded automaton, then $L(M)$ is decidable.

Proof: The number of configurations of M on an input of length n is at most $|Q| \cdot n \cdot |\Gamma|^n$, where $|Q|$ is the number of states and $|\Gamma|$ is the size of the tape alphabet (the tape has at most n symbols).

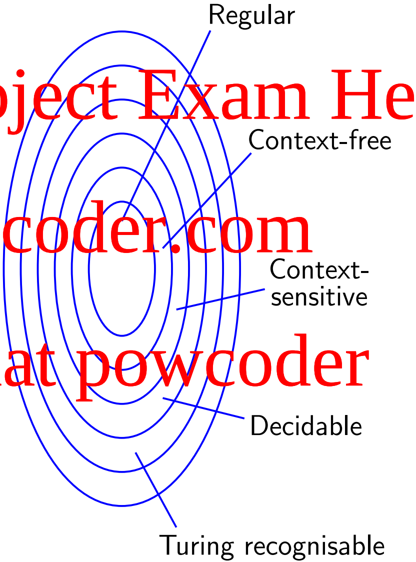
If M accepts w of length n then M does so within at most $|Q| \cdot n \cdot |\Gamma|^n$ steps. Any computation of length more than $|Q| \cdot n \cdot |\Gamma|^n$ is "cycling" and so cannot accept w. If M can't accept w within $|Q| \cdot n \cdot |\Gamma|^n$ steps, it rejects this string.

The Hierarchy of Language Classes

The diagram shows the relations amongst language classes established so far.

But are there Turing recognisable languages that are not decidable?

As it turns out, yes.



Undecidable Problems

An Undecidable Language

Now let us study undecidable problems/languages.

We start by showing that it is undecidable whether a Turing machine accepts a given input string. That is,

$$A_{TM} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$$

is undecidable.

The main difference from the case of A_{CFG}, for example, is that a Turing machine may fail to halt.

TM Acceptance Is Undecidable

Theorem:

$$A_{TM} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$$

is undecidable.

Proof: Assume (for contradiction) that A_{TM} is decided by a TM H:

$$H\langle M, w\rangle = \begin{cases} \mathit{accept} & \text{if } M \text{ accepts } w \\ \mathit{reject} & \text{if } M \text{ does not accept } w \end{cases}$$

Using H we can construct a Turing machine D which decides whether a given machine M fails to accept its own encoding $\langle M\rangle$:

1. Input is $\langle M\rangle$, where M is some Turing machine.
2. Run H on $\langle M, \langle M\rangle\rangle$.
3. If H accepts, reject. If H rejects, accept.

TM Acceptance

In summary:

$$D(\langle M\rangle) = \begin{cases} \textit{accept} & \text{if } M \text{ does not accept } \langle M\rangle \\ \textit{reject} & \text{if } M \text{ accepts } \langle M\rangle \end{cases}$$

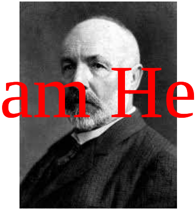
But no machine can satisfy that specification!

Why? Because we obtain an absurdity when we investigate D's behaviour when we run it on its own encoding:

$$D(\langle D\rangle) = \begin{cases} \textit{accept} & \text{if } D \text{ does not accept } \langle D\rangle \\ \textit{reject} & \text{if } D \text{ accepts } \langle D\rangle \end{cases}$$

Hence neither D nor H can exist.

Comparing Sizes of Sets: Cantor's Criterion



So what does 'equals' and 'less' mean for infinite cardinality?

How do we compare the "sizes" of infinite sets?

Cantor's criterion:

- $\mathit{card}(X) \leq \mathit{card}(Y)$ iff there is a total, injective $f : X \to Y$.
- $\mathit{card}(X) = \mathit{card}(Y)$ iff

$$\mathit{card}(X) \leq \mathit{card}(Y) \text{ and } \mathit{card}(Y) \leq \mathit{card}(X).$$

As a consequence, there are (infinitely) many degrees of infinity.

To Infinity and Beyond

X is countable iff $card(X) \leq card(\mathbb{N})$.

X is countably infinite iff $card(X) = card(\mathbb{N})$.

Examples: $\mathbb{Z}$, $\mathbb{N}^k$, and $\mathbb{N}^*$ (the set of all finite sequences of natural numbers) are all countably infinite.

Importantly, Σ^* is countable for all finite alphabets Σ, including the alphabet of printable characters on your keyboard.

$\mathcal{P}(\mathbb{N})$, $\mathbb{N} \to \mathbb{N}$, and $\mathbb{Z} \to \mathbb{Z}$ are uncountable, as can be shown by diagonalisation.

Diagonalisation Showing $\mathbb{Z} \to \mathbb{Z}$ Is Uncountable

Theorem: There is no bijection $h : \mathbb{N} \to (\mathbb{Z} \to \mathbb{Z})$.

Proof: Assume h exists. Then

$$h(0), h(1), h(2), \ldots, h(n), \ldots$$

contains every function in $\mathbb{Z} \to \mathbb{Z}$, without duplicates.

Now construct $f : \mathbb{Z} \to \mathbb{Z}$ as follows:

$$f(n) = h(n)(n) + 1$$

Then $f \neq h(n)$ for all n, so we have a contradiction.

Why This Is Called Diagonalisation

Here is some hypothetical listing of all the functions $h(0), h(1), \ldots$ that make up $\mathbb{Z} \to \mathbb{Z}$:

	0	1	2	3	4	5	…
h(0)	19	3	42	0	7	9	…
h(1)	42	42	42	42	42	42	…
h(2)	42	43	44	45	46	47	…
h(3)	6	93	17	84	6	93	…
h(4)	45	18	-8	-5	63	-9	…
⋮							

Why This Is Called Diagonalisation

Here is some hypothetical listing of all the functions $h(0), h(1), \ldots$ that make up $\mathbb{Z} \to \mathbb{Z}$:

	0	1	2	3	4	5	...
h(0)	19	3	42	0	7	9	...
h(1)	42	42	42	42	42	42	...
h(2)	42	43	44	45	46	47	...
h(3)	6	93	17	84	6	93	...
h(4)	45	18	-8	-5	63	-9	...
⋮							

f is defined in such a way that it cannot possibly be in the listing:

	0	1	2	3	4	5	...
f	20	43	45	85	64	...	...

Algorithms vs Functions

Consider the set of algorithms that realise functions $f : \mathbb{Z} \to \mathbb{Z}$.

How large is that set?

It is infinite, but we can enumerate it. It is contained in Σ^*, where Σ is the set of (printable) characters on my keyboard and that set is countable.

So there cannot be any more, say, Haskell functions, of type `Integer -> Integer` than there are integers. Namely, each Haskell function is represented finitely, as a finite sequence of symbols from a finite alphabet.

Algorithms vs Functions

However, we saw that $\mathbb{Z} \rightarrow \mathbb{Z}$ is not countable.

In other words, there are number-theoretic functions (in fact, lots of them) that do not have a corresponding algorithm.

So are there any "important" functions that are not computable?

As it turns out, yes, very much so!

Problems that Have No Algorithmic Solution

Some undecidable problems:

- Are two given CFGs equivalent?
- Are there strings that a given CFG cannot generate?
- Is a given CFG unambiguous?
- Will a given Python program halt for all input?
- Will a given Java program ever throw a certain exception?

Next week we will explore some other undecidable problems.