22. Computability

Goal: · Formalize the intuitive notion of computable functions:

$$\text{computable} \overset{\text{(Turing)}}{=} \text{computable by a TM.}$$

· Introduce the notions of decidable / semi-decidable properties and recursive / recursively-enumerable sets.

History: · In the 1930s, several algorithms were known to compute certain functions (e.g. Gaussian elimination, Taylor and Newton approximation,

But there was no general definition of being computable.

Only with such a definition it is possible to show that some functions are not computable.

· Turing's definition was successful in that

it actually seems to capture the intuition to being computable.

$$\boxed{\text{This believe is known as Church's thesis. Alonzo Church, 1936.}}$$

The believe is justified by the fact that all models of computation suggested so far have been shown to be covered (and many equivalent to) TMs:

μ-recursive functions (Kurt Gödel 1965, Jacques Herbrand)

λ-calculus (Alonzo Church 1933, Stephen Cole Kleene 1935)

Combinatory logic (Moses Schönfinkel 1924, Haskell B. Curry 1929)

...

The existence of a universal Turing machine (that can simulate any other TM) is another confirmation of Church's thesis.

Phrased differently, computability is not about these models, but about the idea captured by all of them.

22.1 Computable Functions

Intuitively: • We would like to say that a partial function

$$f : \mathbb{N}^k \rightharpoonup \mathbb{N}$$

is computable, if there is an algorithm that

given input $n_1, \ldots, n_k$

↳ stops after finitely many steps

and outputs $f(n_1, \ldots, n_k)$,

in case f is defined on $n_1, \ldots, n_k$, and

↳ does not stop (or can do anything but accept)

if f is undefined on $n_1, \ldots, n_k$.

• In turn, every deterministic algorithm computes a partial function (in the above sense).

Examples:

(1) Input: $n \in \mathbb{N}$

begin while true do

skip;

od

end

The algorithm computes the partial function

$$\Omega : \mathbb{N} \rightharpoonup \mathbb{N}$$
$$n \mapsto \text{undef}$$

that is everywhere undefined.

(2) $f_\pi : \mathbb{N} \rightharpoonup \mathbb{N}$

$$n \mapsto \begin{cases} 1, & \text{if } n \text{ is a prefix of } \pi \\ 0, & \text{otherwise.} \end{cases}$$

For example, $f(314) = 1$ and $f(5) = 0$

The function is computable as we can approximate π precise enough (where precise enough is determined by the length of n).

(3) $g_\pi : \mathbb{N} \rightharpoonup \mathbb{N}$

$$n \mapsto \begin{cases} 1, & \text{if } n \text{ is an infix of } \pi \\ 0, & \text{otherwise.} \end{cases}$$

We do not know this.

May actually be the case: If π is truly random, it will contain every word over $\{0, \ldots, 9\}$ as an infix.

(4) $h_\pi : \mathbb{N} \to \mathbb{N}$

$$n \mapsto \begin{cases} 1, & \text{if } \pi \text{ contains } \underbrace{7\dots7}_{n\text{-times}} \\ 0, & \text{otherwise.} \end{cases}$$

h_π is computable:

Either there are arbitrarily long sequences of 7s in π, or the longest sequence has length $n_0 \in \mathbb{N}$.

- In the first case, we pick

$$h_\pi^1(n) := 1 \qquad \text{for all } n \in \mathbb{N}.$$

- In the second case, we define

$$h_\pi^2(n) := \begin{cases} 1, & \text{if } n \leq n_0 \\ 0, & \text{otherwise.} \end{cases}$$

This variant of h_π is computable as $n \leq n_0$ can be decided by an algorithm.

- One of the cases has to apply, so there definitely is an algorithm.

We just do not know which one is the right one.

But this is not required for computability: There only has to be an algorithm.

If we know the algorithm, we would say a function is effectively computable.

(5) $lba : \mathbb{N} \to \mathbb{N}$

$$n \mapsto \begin{cases} 1, & \text{if } NLBA = DLBA \\ 0, & \text{otherwise} \end{cases}$$

lba is computable:

Algorithm 1: $lba^1(n) := 1$, if Kuroda's first problem has a positive answer

Algorithm 2: $lba^2(n) := 0$, otherwise.

Again we do not know which algorithm to apply,
but it is one of the two.

(6) Is a function similar to f_π (a function that approximates the value) computable for every real number?

No! There are uncountably many reals
but countably many programs / TMs
(we can encode them as strings over $\{0,1\}$,
see next section).
But every number requires its own program.

In this sense, there are computable and uncomputable numbers
(for example, every rational number is computable).

We now modify the definition of Turing machines
to compute functions rather than accept languages.

Definition:

- A partial function $f: \mathbb{N}^k \rightharpoonup \mathbb{N}$ is (Turing) computable,
if there is a DTM $M = (Q, \Sigma, \Gamma, q_0, \sqcup, \delta, Q_F)$
so that for all $n_1, \dots, n_k \in \mathbb{N}$:

$$f(n_1, \dots, n_k) = m$$

iff $q_0 \, bin(n_1) \# \dots \# bin(n_k) \longrightarrow^* \sqcup \dots \sqcup q_f \, bin(m) \sqcup \dots \sqcup$,
where $q_f \in Q_F$ and $bin(n)$ is the binary representation of n (without leading 0s).

- Computability for $f: \Sigma^* \rightharpoonup \Sigma^*$ is defined similarly.

Recall: • Every non-deterministic Turing machine can be turned into a deterministic one.
Hence, referring to a DTM is no restriction (and natural for functions).

- We can even assume the DTM no longer changes anything (neither the tape nor the head nor its state) once it entered a final state.

Remark:

If $f: \mathbb{N}^k \to \mathbb{N}$ is undefined on an input, the corresponding machine will not reach a configuration of the given form:

↳ it will get stuck (not possible if we assume determinism),

↳ it will loop indefinitely, or

↳ it will end in a configuration that is not of the given form.

Example:

The above functions sb, f_π, h_π, and lba are all computable.

To show that there is a function that is not computable, recall the definition of countability.

Definition:

A set A is countable, if $A = \emptyset$ or there is a surjective function $f: \mathbb{N} \to A$.

Phrased differently, there is an enumeration $f(0), f(1), f(2), \ldots$ (that is not necessarily implementable) that lists every element of A.

Note that $f(i) = f(j)$ for $i \neq j$ is possible (need not be injective).

Theorem:

There is a function $f: \mathbb{N} \to \mathbb{N}$ that is not computable.

Proof: · Use the same idea on uncountably many functions and countably many programs / TMs.

· Towards a contradiction, assume every function $f : \mathbb{N} \to \mathbb{N}$ is computable by a TM M_f.

Since there are only countably many TMs, this means the set $\mathcal{F}$ of all total functions $f : \mathbb{N} \to \mathbb{N}$ is countable.

Hence, there is a surjective function $c : \mathbb{N} \to \mathcal{F}$.

· We define $g : \mathbb{N} \to \mathbb{N}$ by

$$g(n) := f_n(n) + 1, \quad \text{where } f_n = c(n).$$

· Since c is surjective, there is a value i so that

$$g = c(i).$$

But then

$$\begin{aligned} g(i) &= f_i(i) + 1 \\ &= (c(i))(i) + 1 \\ &= g(i) + 1. \end{aligned}$$

The equation implies $1 = 0$ ↯.

Hence, the assumption that every function $f : \mathbb{N} \to \mathbb{N}$ is computable has to be false. □

Illustration:

$f_n(i)$:

$n \backslash i$	$f_n(0)$	$f_n(1)$	$f_n(2)$	
0	8 9	3	20	f_0 represented as a sequence
1	1	100 101	23	f_1 represented as a sequence
2	205	1	110 111	f_2 represented as a sequence

9

Function g is defined by

- taking the diagonal
- and adding 1:

$$g(0) = 9, \quad g(1) = 101, \quad g(2) = 111, \ldots$$

The point in the definition of g is that the function is different from all f_i.

This is a diagonalization proof.

22.2 Decidability

Goal: Introduce notions of computability that are tailored towards languages (sets / problems).

Definition:

- A set $A \subseteq \Sigma^*$ (or $A \subseteq \mathbb{N}$) is decidable, if the (total) characteristic function $\chi_A : \Sigma^* \to \{0,1\}$ is computable.
  Remember, $\chi_A(w) := \begin{cases} 1, & \text{if } w \in A \\ 0, & \text{otherwise.} \end{cases}$
- A set $A \subseteq \Sigma^*$ is semi-decidable, if the partial/half characteristic function $\chi'_A : \Sigma^* \nrightarrow \{0,1\}$ is computable.
  Here, $\chi'_A(w) := \begin{cases} 1, & \text{if } w \in A \\ \text{undef,} & \text{otherwise.} \end{cases}$

Language $A = \{ w \in \Sigma^* \mid$ the condition defining $A \}$

are often called <u>decision problems</u> and written as

<u>Given</u>: $w \in \Sigma^*$.

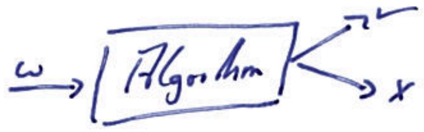
<u>Question</u>: Does the condition defining A hold for w?

Because we actually check a condition, some books use decidable / semi-decidable for properties (conditions / predicates).

<u>Illustration</u>:

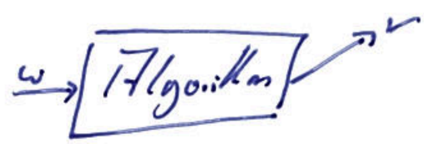
<u>Decidability</u>:

There is an algorithm (DTM or NTM) that terminates on every input and gives the correct answer.



<u>Semi-decidability</u>:

There is an algorithm that only stops properly on yes-instances. It may loop forever, get stuck, or halt in an inappropriate configuration otherwise. Note that in a loop we are not sure whether termination will still follow.



<u>Example</u>:

Every context-sensitive language $L(G)$ is decidable.

<u>Proof</u>:

We called this MEMBERSHIP $L(G)$ and gave a decision procedure in Section 14.

<u>Theorem</u>:

A language $A \subseteq \Sigma^*$ is decidable iff both A and $\overline{A}$ are semi-decidable.

Proof: „⇒" ✓

„⇐" Let M_A be the semi-decision procedure for A
and $M_{\overline{A}}$ be the semi-decision procedure for $\overline{A}$.
We construct the following algorithm as a decision procedure for A.
Note that we rely on Church's thesis to turn it into a TM:

<u>Input</u>: $w \in \Sigma^*$.
<u>begin</u> <u>for</u> $i = 1, 2, \dots$ <u>do</u>
  <u>if</u> M_A accepts w in i steps <u>then</u>
   <u>output</u> 1;
  <u>else if</u> $M_{\overline{A}}$ accepts w in i steps <u>then</u>
   output 0;
  fi
 <u>od</u>
<u>end</u>

$\square$

There is a different point of view to computability:
Rather than <u>deciding</u> whether a given word is in the language
let us <u>enumerate</u> the elements of the language.

<u>Definition</u>:

- A language $A \subseteq \Sigma^*$ is <u>recursively enumerable</u>, if $A = \emptyset$ or there is a (total and) <u>computable</u> function
$$f: \mathbb{N} \to \Sigma^*$$
so that
$$A = \{ f(0), f(1), f(2), \dots \}.$$
(Note that $f(i) = f(j)$ for $i \neq j$ is possible.)
We say that <u>f enumerates A</u>.
- A language $A \subseteq \Sigma^*$ is <u>recursive</u>, if A and $\overline{A}$ are recursively enumerable.

People use recursive / recursively enumerable for languages whereas decidable / semi-decidable is used for properties. Yet, the distinction is artificial and does not matter.

<u>Theorem:</u>

A language $A \subseteq \Sigma^*$ is recursively enumerable iff it is semi-decidable.

<u>Proof:</u>

"$\Rightarrow$" Let A be recursively enumerable due to the (total and) computable function $f: \mathbb{N} \to \Sigma^*$.

The following is a semi-decider for A:

```
input: w ∈ Σ*.
begin for i = 0,1,2,... do
        if f(i) = w then
          output 1;
        fi
      od
end
```

"$\Leftarrow$" We need a function that takes a single element $n \in \mathbb{N}$ and yields an element $w \in A$.

To obtain w, n will represent a pair (word, steps).

The idea is <u>dovetailing</u> (from cards ):

↳ Enumerate words $w_0, w_1, w_2, \ldots$ in Σ^*.

↳ Simulate steps $0, 1, 2, \ldots$.

Let M be an algorithm that semi-decides A.

The following algorithm lists the elements of A:

```
begin for i = 0,1,2,... do
        if M accepts w_i in ≤ i steps do
          print w_i
        fi
      od
end
```

The function $f: \mathbb{N} \to \Sigma^*$ that we need outputs, on input n, the n-th element in the list produced by the previous algorithm.
This just needs one more counter.

Since M is a semi-decider for A, we only obtain words in the language.
In turn, if $w \in A$, then M accepts w after some, say k, steps.
So w occurs in the printed list and hence occurs as a function value for some $n \in \mathbb{N}$. $\square$

__Corollary:__
A language $A \subseteq \Sigma^*$ is recursive iff it is decidable.

__Summary:__
The following are all equivalent for $A \subseteq \Sigma^*$:

- A is recursively enumerable
- A is semi-decidable
- $A = L(M)$ for M a TM
- A is of type-0
- χ'_A is computable
- A is the range of a total computable function $f: \mathbb{N} \to \Sigma^*$
- A is the domain of definition of a partial computable function $g: \Sigma^* \to \Gamma^*$

$$w \mapsto \begin{cases} \varepsilon, & \text{if } w \in A \\ \text{undef}, & \text{otherwise} \end{cases}$$

Comment:

We comment on the difference between countable and recursively-enumerable sets.

- Note that every subset A' of a countable set

$$A = \{f(0), f(1), f(2), \dots\}$$

is again countable.

Let $a \in A' \neq \emptyset$ (empty sets are countable by definition).

We define

$$g(n) := \begin{cases} f(n), & \text{if } f(n) \in A' \\ a, & \text{otherwise.} \end{cases}$$

Then

$$A' = \{g(0), g(1), g(2), \dots\}.$$

- It is not true that every subset of a recursively-enumerable set is again recursively enumerable.

Let

$$f : \mathbb{N} \to \Sigma^*$$

be a total and computable function that recursively enumerates all Turing machines (encoded as words, see next section):

$$TMs = \{f(0), f(1), f(2), \dots\}.$$

Take the subset

$$UnivDivTMs \subseteq TMs$$

of Turing machines that diverge on all inputs.

(Note that they correspond to the programs c with $\vdash \{true\} c \{false\}$.)

This set is known to be <u>not</u> recursively enumerable.

We now study in more detail which problems cannot be solved algorithmically.