Decidability and Computability

Review: A language is
- recognisable (or semi-decidable, or recursively enumerable) if it is the language of all words recognised by some Turing machine
- decidable (or recursive) if it is the language of a Turing machine that always halts, even on inputs that are not accepted
- undecidable if it is not decidable

Instead of acceptance of words, we can also consider other computations:

**Definition 3.1:** A TM $M$ computes a partial function $f_M : \Sigma^* \rightarrow \Sigma^*$ as follows. We have $f_M(w) = v$ for a word $w \in \Sigma^*$ if $M$ halts on input $w$ with a tape that contains only the word $v \in \Sigma^*$ (followed by blanks).

In this case, the function $f_M$ is called computable. Total, computable functions are called recursive.

Functions may therefore be computable or uncomputable.

Undecidability is Real

A fundamental insight of computer science and mathematics is that there are undecidable languages:

**Theorem 3.2:** There are undecidable languages over every alphabet $\Sigma$.

**Proof:** See exercise. □

Analogously, there are uncomputable functions.

Unknown ≠ Undecidable

How do we find concrete undecidable problems?

It is not enough to not know how to solve a problem algorithmically!

**Example 3.3:** Let $L_\pi$ be the set of all finite number sequences, that occur in the decimal representation of $\pi$. For example, $14159265 \in L_\pi$ and $41 \in L_\pi$.

We do not know if the language $L_\pi$ is decidable, but it might be (e.g., if every finite sequence of digits occured in $\pi$, which, however, is not known to be true today).
There are even cases where we are sure that a problem is decidable without knowing how to solve it.

**Example 3.4 (after Uwe Schöning):** Let \( L_{7^n} \) be the set of all number sequences of the form \( 7^n \) that occur in the decimal representation of \( \pi \).

\( L_{7^n} \) is decidable:

- **Option 1:** \( \pi \) contains sequences of arbitrary many \( 7 \). Then \( L_{7^n} \) is decided by a TM that accepts all words of the form \( 7^n \).
- **Option 2:** \( \pi \) contains sequences of \( 7 \)'s only up to a certain maximal length \( \ell \). Then \( L_{7^n} \) is decided by a TM that accepts all words of the form \( 7^n \) with \( n \leq \ell \).

In each possible case, we have a practical algorithm – we just don’t know which one is correct.

---

**A First Undecidable Problem (1)**

**Question:** If a TM halts, how long may this take in the worst case?

**Answer:** Arbitrarily long, since:

- (a) the input might be arbitrarily long
- (b) the TM can be arbitrarily large

**Question:** If a TM with \( n \) States and a two-element tape alphabet \( \Gamma = \{x, □\} \) halts on the empty input tape, how long may this take in the worst case?

**Answer:** That depends on \( n \) . . .

**Definition 3.5:** We define \( S(n) \) as the largest number of steps that any DTM with \( n \) states and tape alphabet \( \Gamma = \{x, □\} \) executes on the empty tape, before it eventually halts.

**Observation:** \( S \) is well defined.

- The number of TMs with at most \( n \) states is finite
- Among the relevant \( n \)-state TMs there must be a largest number of steps before halting (TMs that do not halt are ignored)

---

**Busy Beaver**

A small variation of the step counter function leads to the Busy-Beaver Problem:

**Definition 3.6:** The Busy-Beaver function \( \Sigma: \mathbb{N} \rightarrow \mathbb{N} \) is a total function, where \( \Sigma(n) \) is the maximal number of \( x \) that a DTM with at most \( n \) states and tape alphabet \( \Gamma = \{x, □\} \) can write when starting on the empty tape and before it eventually halts.

**Note:** The exact value of \( \Sigma(n) \) depends on details of the TM definition.

Most works in this area assume a two-sided infinite tape that can be extended to the left and to the right if necessary.

---

**Example**

The Busy-Beaver number \( \Sigma(2) \) is 4 when using a two-way infinite tape.

The following TM implements this behaviour:

\[
\begin{align*}
\square & \mapsto x, R \\
x & \mapsto x, L
\end{align*}
\]

We obtain:

\[
A \quad □ \quad x \quad B \\
\rightarrow \\
\rightarrow
\]

\[
\begin{align*}
\square & \mapsto x, L \\
x & \mapsto x, x \\
A & \quad □ \quad x
\end{align*}
\]

\[
\begin{align*}
A \quad □ \quad x \quad B \quad □ \quad x \\
\rightarrow \\
\rightarrow
\end{align*}
\]

\[
\begin{align*}
A & \quad □ \quad x \quad x \quad x \\
\rightarrow \\
\rightarrow
\end{align*}
\]

\[
\begin{align*}
A & \quad □ \quad x \quad x \quad x \quad x \\
B & \quad □ \quad x
\end{align*}
\]
Computing Busy-Beaver?

How hard could this possibly be?

**Theorem 3.7:** The Busy-Beaver function is not computable.

**Proof sketch:** Suppose for a contradiction that $\Sigma$ is computable.

- Then we can define a TM $M_{\Sigma}$ with tape alphabet $\{x, \square\}$ that computes $x^n \mapsto x^{\Sigma(n)}$.
- Let $M_{+1}$ be a TM that computes $x^n \mapsto x^{n+1}$.
- Let $M_{x^2}$ be a TM that computes $x^n \mapsto x^{2n}$.
- Let $k$ be the total number of states in $M_{\Sigma}$, $M_{+1}$, and $M_{x^2}$. There is a TM $I_k$ with $k$ states that writes the word $x^k$ to the empty tape.
- When executing $I_k$, $M_{x^2}$, $M_{\Sigma}$, and $M_{+1}$ after another, the result is a TM with $2k$ states that writes $\Sigma(2k) + 1$ times $x$ before halting.
- Hence $\Sigma(2k) \geq \Sigma(2k) + 1$ – contradiction.

**Note 1:** The proof involves an interesting idea of using TMs as “sub-routines” in other TMs. We will use this again later on.

**Note 2:** If a TM can compute $f : \mathbb{N} \rightarrow \mathbb{N}$ in the usual unary encoding, it is not hard to get a TM for $x^n \mapsto x^{f(n)}$ by just using unary encoding instead.

**Note 3:** Transforming an arbitrary TM into one that uses only symbols $\{x, \square\}$ on its tape is slightly more involved, but doable.

Busy Beaver in Practice

“Maybe the theoretical uncomputability is not really relevant after all – in practice, we surely can find values for practically relevant sizes of TMs, no?”

Well, progress since the 1960s has been rather modest:

<table>
<thead>
<tr>
<th>$n$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Sigma(n)$</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>13</td>
<td>$\geq 4098$</td>
<td>$\geq 3.5 \times 10^{18267}$</td>
<td>gigantic</td>
<td>insane</td>
</tr>
</tbody>
</table>

For $n = 10$, one has found a lower bound of the form $\Sigma(10) > 3^{3^{3^{\cdot^{\cdot^3}}}}$, where the complete expression has more than $7.6 \times 10^{12}$ occurrences of the number 3.
The Universal Machine

A first important observation of Turing was that TMs are powerful enough to simulate other TMs:

Step 1: Encode Turing Machines $M$ as words $\langle M \rangle$

Step 2: Construct a universal Turing Machine $U$, which gets $\langle M \rangle$ as input and then simulates $M$

Step 2: The Universal Turing Machine

We define the universal TM $U$ as multi-tape TM:

Tape 1: Input tape of $U$: contains $\langle M \rangle$ as input
Tape 2: Working tape of $U$
Tape 3: Stores the state of the simulated TM
Tape 4: Working tape of the simulated TM

The working principle of $U$ is easily sketched:

- $U$ validates the input, copies $\langle w \rangle$ to Tape 4, moves the head on Tape 4 to the start and initialises Tape 3 with $\text{bin}(0)$ (i.e., $\langle q_0 \rangle$).
- In each step $U$ reads an (encoded) symbol from the head position on Tape 4, and searches for the simulated state (Tape 3) a matching transition in $\langle M \rangle$ on Tape 1 (w.l.o.g. assume that the final states of the encoded TM have no transitions):
  - Transition found: update state on Tape 3; replace the encoded symbol on Tape 4 by the new symbol; move the head on Tape 4 accordingly
  - Transition not found: if the state on Tape 3 is $q_{\text{accept}}$, then go to the final accepting state; else go to the final rejecting state

Step 1: encoding Turing Machines

Any reasonable encoding of a TM $M = \langle Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}} \rangle$ is usable, e.g., the following (for DTMs):

- We use an alphabet $\{0, 1, \#\}$
- States are enumerated in any order (beginning with $q_0$), and encoded in binary:
  $$Q = \{q_0, \ldots, q_n\} \leadsto \langle Q \rangle = \text{bin}(0)\# \cdots \#\text{bin}(n)$$
- We also encode $\Gamma$ and the directions $[R, L]$ in binary
- A transition $\delta(q_i, \sigma_i) = (q_j, \sigma_j, D)$ is encoded as 5-tuple:
  $$\text{enc}(q_i, \sigma_i) = \text{bin}(i)\#\text{bin}(n)\#\text{bin}(j)\#\text{bin}(m)\#\text{bin}(D)$$
- The transition function is encoded as a list of all these tuples, separated with $\#$:
  $$\langle \delta \rangle = (\text{enc}(q_i, \sigma_i)\#)_{i \in \Gamma, \sigma_i \in \Sigma}$$
- Combining everything, we set $\langle M \rangle = \langle Q \rangle\#\langle \Sigma \rangle\#\langle \Gamma \rangle\#\langle \delta \rangle\#\langle q_{\text{accept}} \rangle\#\langle q_{\text{reject}} \rangle$

The Theory of Software

Theorem 3.8: There is a universal Turing Machine $U$, that, when given an input $\langle M \rangle\#\langle w \rangle$, simulates the behaviour of a DTM $M$ on $w$:

- If $M$ halts on $w$, then $U$ halts on $\langle M \rangle\#\langle w \rangle$ with the same result
- If $M$ does not halt on $w$, then $U$ does not halt on $\langle M \rangle\#\langle w \rangle$ either

Our construction is for DTMs that recognise languages (“Turing acceptors”) – DTMs that compute partila functions can be simulated in a similar fashion.

Practical consequences:

- Universal computers are possible
- We don’t have to buy a new computer for every application
- Software exists
The Halting Problem

A classical undecidable problem:

**Definition 3.9:** The Halting Problem consists in the following question:
Given a TM $M$ and a word $w$, will $M$ ever halt on input $w$?

We can formulate the Halting Problem as a word problem by encoding $M$ and $w$:

**Definition 3.10:** The Halting Problem is the word problem for the language $P_{halt} = \{\langle M \rangle \# \# \langle w \rangle | M \text{ halts on input } w \}$,
where $\langle M \rangle$ and $\langle w \rangle$ are suitable encodings of $M$ and $w$, for which $\#\#$ can be used as separator.

**Remark:** Wrongly encoded inputs are rejected.

''Proof'' by Intuition

**Theorem 3.11:** The Halting Problem $P_{halt}$ is undecidable.

**Proof:** The opposite would be too good to be true. Many unsolved problems could then be solved immediately.

**Example 3.12:** Goldbach’s Conjecture (Christian Goldbach, 1742) states that every even number $n \geq 4$ is the sum of two primes. For instance, $4 = 2 + 2$ and $100 = 47 + 53$.

On can easily give an algorithm $A$ that verifies Goldbach’s conjecture systematically by testing it for every even number starting with 4:
- Success: Test the next even number
- Failure: Terminate with output “Goldbach was wrong!”

The question “Will $A$ halt?” therefore is equivalent to the question “Is Goldbach’s conjecture wrong?”

Many other important open problems could be solved in this way.

Proof by “Diagonalisation”

**Theorem 3.11:** The Halting Problem $P_{halt}$ is undecidable.

**Proof:** By contradiction: Suppose there is a decider $H$ for the Halting Problem.

Then one can construct a TM $D$ that does the following:

1. Check if the given input is a TM encoding $\langle M \rangle$
2. Simulate $H$ on input $\langle M \rangle \# \# \langle \langle M \rangle \rangle$, that is, check if $M$ halts on $\langle M \rangle$
3. If yes, enter an infinite loop; if no, halt and accept

Will $D$ accept the input $\langle D \rangle$?

$D$ halts and accepts if and only if $D$ does not halt

Contradiction. □
Proof by Reduction

**Theorem 3.11:** The Halting Problem $P_{\text{Halt}}$ is undecidable.

**Proof:** Suppose that the Halting Problem is decidable.

An algorithm:

- Input: natural number $k$ (in binary)
- Iterate over all Turing machines $M$ that have $k$ states and tape alphabet $\{x, \square\}$:
  - Decide if $M$ halts on the empty input $\varepsilon$ (possible if the Halting problem is decidable)
  - If yes, then simulate $M$ on the empty input and, when $M$ has halted, count the number of $x$ on the tape (possible, since there are universal TMs)
- Output: the maximal number of $x$ written.

This algorithm would compute the Busy-Beaver function $\Sigma : \mathbb{N} \to \mathbb{N}$.

We have already shown that this is impossible – contradiction. □

Turing Reductions

Our previous proof constructs an algorithm for one task (Busy Beaver) by calling subroutines for another task (the Halting Problem).

This idea can be generalised:

**Informal Definition 3.13:** A problem $P$ is Turing reducible to a problem $Q$ (in Symbols: $P \leq_T Q$), if $P$ can be solved by a program that may call $Q$ as a sub-program.

**Example 3.14:** Our proof uses a reduction of the Busy-Beaver computation to the Halting problem. Note that the subroutine might be called exponentially many times here.

To make this more formal, we need oracles.

Oracles

**Definition 3.15:** An Oracle Turing Machine (OTM) is a Turing machine $M$ with a special tape, called the oracle tape, and distinguished states $q_?$, $q_{\text{yes}}$, and $q_{\text{no}}$. For a language $O$, the oracle machine $M^O$ can, in addition to the normal TM operations, do the following:

Whenever $M^O$ reaches $q_?$, its next state is $q_{\text{yes}}$ if the content of the oracle tape is in $O$, and $q_{\text{no}}$ otherwise.

- The word problem for $O$ might be very hard or even undecidable
- Nevertheless, asking the oracle always takes just one step
- For dramatic effect, we might assert that the contents of the oracle tape is consumed (emptied) during this mysterious operation. However, this does not usually make a difference to our results.

**Definition 3.16:** A problem $P$ is Turing reducible to a problem $Q$ (in Symbols: $P \leq_T Q$), if $P$ is decided by an OTM $M^O$ with oracle $Q$.

Undecidability via Turing Reductions

One can use Turing reductions to show undecidability:

**Theorem 3.17:** If $P$ is undecidable and $P \leq_T Q$, then $Q$ is undecidable.

**Proof:** Via contrapositive: If $P \leq_T Q$ and $Q$ is decidable, then we can implement the OTM that shows $P \leq_T Q$ as a regular TM, which shows that $P$ is decidable. □

Here is a small application:

**Theorem 3.18:** The language $P_{\text{Halt}} = \{\langle M\rangle \#\#(w) \mid M \text{ does not halt on } w\}$ (the “Non-Halting Problem”) is undecidable.

**Proof sketch:** Decide Halting by using $P_{\text{Halt}}$ as an oracle and inverting the result. Check TM encoding first (wrong encodings are rejected by Halting and Non-Halting). □
Special cases of the Halting Problem are usually not simpler:

**Definition 3.19:** The ε-Halting Problem consists in the following question: Given a TM \( M \), will \( M \) ever halt on the empty input \( \varepsilon \)?

**Theorem 3.20:** The ε-Halting Problem is undecidable.

**Proof:** We define an oracle machine for deciding Halting:

- Input: A Turing machine \( M \) and a word \( w \).
- Construct a TM \( M_w \) that runs in two phases:
  1. Delete the input tape and write the word \( w \) instead
  2. Process the input like \( M \)
- Solve the ε-Halting problem for \( M_w \) (oracle).
- Output: output of the ε-Halting Problem

This Turing-reduces Halting to ε-halting, so the latter is also undecidable.

---

**Summary and Outlook**

Busy Beaver is uncomputable

Halting is undecidable (for many reasons)

Oracles and Turing reductions formalise the notion of a “subroutine” and help us to transfer our insights from one problem to another

**What’s next?**

- Some more undecidability
- Recursion and self-referentiality
- Actual complexity classes