TECHNISCHE UNIVERSITÄT DRESDEN

COMPLEXITY THEORY

Lecture 3: Undecidability

Markus Krötzsch
Knowledge-Based Systems

TU Dresden, 17th Oct 2017

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 allways halts, even on inputs that are not accepted
- undedicable if it is not decidable

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

Definition 3.1: A TM $\mathcal{M}$ computes a partial function $f_\mathcal{M} : \Sigma^* \to \Sigma^*$ as follows. We have $f_\mathcal{M}(w) = v$ for a word $w \in \Sigma^*$ if $\mathcal{M}$ halts on input w with a tape that contains only the word $v \in \Sigma^*$ (followed by blanks).
In this case, the function $f_\mathcal{M}$ 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 Σ.

Proof: See exercise. □

Analoguously, 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 $\mathbf{L}_\pi$ be the set of all finite number sequences, that occur in the decimal representation of π. For example, `14159265` $\in \mathbf{L}_\pi$ and `41` $\in \mathbf{L}_\pi$.

We do not know if the language $\mathbf{L}_\pi$ is decidable, but it might be (e.g., if every finite sequence of digits occured in π, which, however, is not known to be true today).

Unknown ≠ Undecidable (2)

There are even case, where we are sure that a problem is decidable without knowning how to solve it.

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

$\mathbf{L}_{\pi 7}$ is decidable:

- Option 1: π contains sequences of arbitrary many 7. Then $\mathbf{L}_{\pi 7}$ is decided by a TM that accepts all words of the form 7^n.
- Option 2: π contains sequences of 7s only up to a certain maximal length ℓ. Then $\mathbf{L}_{\pi 7}$ 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 $\Gamma = \{\mathtt{x}, \square\}$ 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 $\Gamma = \{\mathtt{x}, \square\}$ 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

Tibor Radó, BB inventor

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

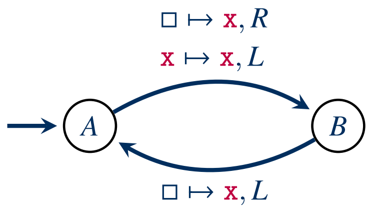
Definition 3.6: The Busy-Beaver function $\mathbf{\Sigma} : \mathbb{N} \to \mathbb{N}$ is a total function, where $\mathbf{\Sigma}(n)$ is the maximal number of $\mathtt{x}$ that a DTM with at most n states and tape alphabet $\Gamma = \{\mathtt{x}, \square\}$ can write when starting on the empty tape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

Note: The exact value of $\mathbf{\Sigma}(n)$ 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 $\mathbf{\Sigma}(2)$ is 4 when using a two-way infinite tape.
The following TM implements this behaviour:



We obtain: $A\square \vdash \mathtt{x}B\square \vdash A\mathtt{xx} \vdash B\square\mathtt{xx} \vdash A\square\mathtt{xxx} \vdash \mathtt{x}B\mathtt{xxx}$

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 $\mathbf{\Sigma}$ is computable.

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

Proof Notes

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 $f : \mathbb{N} \to \mathbb{N}$ in the usual inary encoding, it is not hard to get a TM for $\mathtt{x}^n \mapsto \mathtt{x}^{f(n)}$ by just using unary encoding instead.

Note 3: Transforming an arbitrary TM into one that uses only symbols $\{\mathtt{x}, \square\}$ 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:

n:	1	2	3	4	5	6	7	8
$\mathbf{\Sigma}(n)$:	1	4	6	13	≥ 4098	$\geq 3,5 \times 10^{18267}$	gigantic	insane

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

Universality

The Universal Machine

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

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

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

Step 1: encoding Turing Machines

Any reasonable encxoding of a TM $\mathcal{M} = \langle Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}}\rangle$ 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: $Q = \{q_0, \ldots, q_n\} \rightsquigarrow \langle Q\rangle = \text{bin}(0)\#\cdots\#\text{bin}(n)$
- We also encode Γ and the directions $\{R, L\}$ in binary
- A transition $\delta(q_i, \sigma_n) = \langle q_j, \sigma_m, D\rangle$ is encoded as 5-tuple: $\text{enc}(q_i, \sigma_n) = \text{bin}(i)\#\text{bin}(n)\#\text{bin}(j)\#\text{bin}(m)\#\text{bin}(D)$
- The transition function is encoded as a list of all these tuples, separated with #: $\langle\delta\rangle = (\text{enc}(q_i, \sigma_n)\#)_{q_i\in Q, \sigma_i\in\Gamma}$
- Combining everything, we set $\langle\mathcal{M}\rangle = \langle Q\rangle\#\#\langle\Sigma\rangle\#\#\langle\Gamma\rangle\#\#\langle\delta\rangle\#\#\langle q_{\text{accept}}\rangle\#\#\langle q_{\text{reject}}\rangle$

We can also encode arbitrary words to match this encoding:

- For a word $w = a_1 \cdots a_\ell$ we define $\langle w\rangle = \text{bin}(a_1)\#\cdots\#\text{bin}(a_\ell)$

Step 2: The Universal Turing Machine

We define the universal TM $\mathcal{U}$ as multi-tape TM:

Tape 1: Input tape of $\mathcal{U}$: contains $\langle\mathcal{M}\rangle\#\#\langle w\rangle$

Tape 2: Working tape of $\mathcal{U}$

Tape 3: Stores the state of the simulated TM

Tape 4: Working tape of the simulated TM

The working principle of $\mathcal{U}$ is easily sketched:

- $\mathcal{U}$ validates the inpurt, 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 $\mathcal{U}$ reads an (encoded) symbol from the head position on Tape 4, and searches for the simulated state (Tape 3) a matching transition in $\langle\mathcal{M}\rangle$ 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_{accept}, then go to the final accepting state; else go to the final rejecting state

The Theory of Software

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

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

Undecidable Problems and Reductions

The Halting Problem

A classical undecidable problem:

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

We can fomulate the Halting Problem as a word problem by encoding $\mathcal{M}$ and w:

Definition 3.10: The Halting Problem is the word problem for the language

$$\mathbf{P}_{\mathsf{Halt}} = \{\langle\mathcal{M}\rangle\#\#\langle w\rangle \mid \mathcal{M} \text{ halts on input } w\},$$

where $\langle\mathcal{M}\rangle$ und $\langle w\rangle$ are suitable encodings of $\mathcal{M}$ and w, for which ## can be used as separator.

Remark: Wrongly encoded inputs are rejected.

"‘Proof"’ by Intuition

Theorem 3.11: The Halting Problem $\mathbf{P}_{\mathsf{Halt}}$ 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 $\mathcal{A}$ 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 $\mathcal{A}$ halt?” therefore is equivalent ot 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 $\mathbf{P}_{\mathsf{Halt}}$ is undecidable.

Proof: By contradiction: Suppose there is a decider $\mathcal{H}$ for the Halting Problem.

Then one can construct a TM $\mathcal{D}$ that does the following:

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

Will $\mathcal{D}$ accept the input $\langle\mathcal{D}\rangle$?

$\mathcal{D}$ halts and accepts if and only if $\mathcal{D}$ does not halt

Contradiction. □

Proof by Reduction

Theorem 3.11: The Halting Problem $\mathbf{P}_{\text{Halt}}$ is undecidable.

Proof: Suppose that the Halting Problem is decidable.

An algorithm:

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

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

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: $\mathbf{P} \leq_T \mathbf{Q}$), 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 orcales.

Oracles

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

Whenever $\mathcal{M}^{\mathbf{O}}$ reaches $q_?$, its next state is q_{yes} if the content of the oracle tape is in **O**, and q_{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: $\mathbf{P} \leq_T \mathbf{Q}$), if **P** is decided by an OTM $\mathcal{M}^{\mathbf{Q}}$ with oracle **Q**.

Undecidability via Turing Reductions

One can use Turing recductions to show undecidability:

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

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

Here is a small application:

Theorem 3.18: The language $\mathbf{P}_{\overline{\text{Halt}}} = \{\langle\mathcal{M}\rangle\#\#\langle w\rangle \mid \mathcal{M} \text{ does not halt on } w\}$ (the "Non-Halting Problem") is undecidable.

Proof sketch: Decide Halting by using $\mathbf{P}_{\overline{\text{Halt}}}$ as an oracle and inverting the result. Check TM encoding first (wrong encodings are rejected by Halting and Non-Halting). □

ε-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 $\mathcal{M}$,
will $\mathcal{M}$ ever halt on the empty input ε?

Theorem 3.20: The ε-Halting Problem is undecidable.

Proof: We define an oracle machine for deciding Halting:

- Input: A Turing machine $\mathcal{M}$ and a word w.
- Construct a TM $\mathcal{M}_w$ that runs in two phases:
  (1) Delete the input tape and write the word w instead
  (2) Process the input like $\mathcal{M}$
- Solve the ε-Halting problem for $\mathcal{M}_w$ (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)

Orcales 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