

Undecidability

Hilbert's 10th Problem: Give an algorithm that given a polynomial decides if the polynomial has integer roots or not.

The problem was posed in 1900. In 1970 it was proved that there can be no such algorithm.

Already in the 1930s several problems had been proved to be unsolvable. One example is the Halting Problem.

We will talk about undecidable problems. It means that the problem cannot be decided by an algorithm (more precise definition later). But then, what is an algorithm?

We could replace algorithms with Computers or Programs.
But we will replace it with Turing Machines.

A Turing Machine is a very primitive type of computer. A definition and description of Turing Machines will be given in the next lecture.

Why Turing Machines?

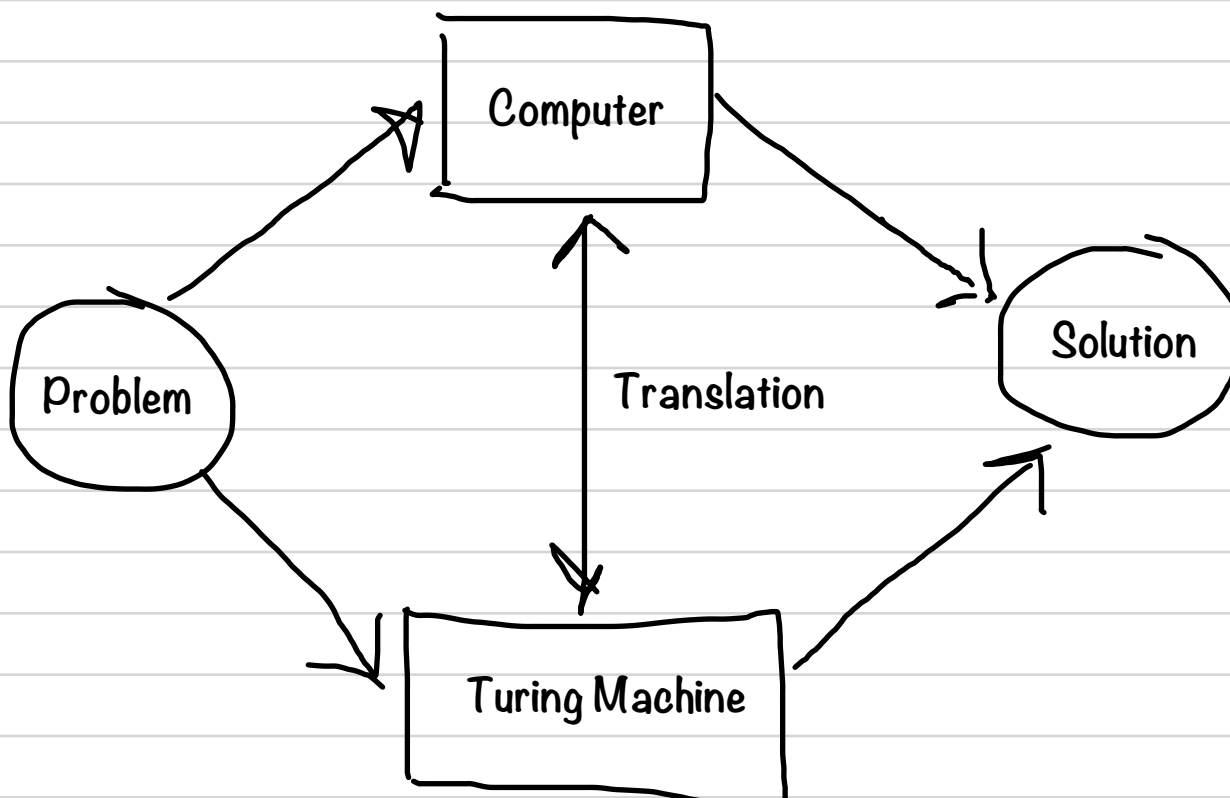
Why do we want to use Turing Machines to solve problems?

The idea is that since they are so simple it is more easy to decide what they can do or not than it would be for more complex computers.

Then we use the famous Church-Turing's Thesis.

Church-Turing (one form): If there is a problem A that can be decided by a computer then it can also be decided by a Turing Machine.

A rough sketch:



We will now give a presentation of uncomputability and undecidability. Usually these concepts are defined and analysed with the Turing Machines. But we can replace Turing Machines with programs written in some language. What language? Well, it doesn't matter. We will just need some facts about programs.

1. Every program can be described by its code.
2. It is possible to enumerate the set of all program codes.
(Lexicographically for instance.)
3. This means that there are only countably many possible programs.

Given a program P , the code $c(P)$ of P is a string. Sometimes it is convenient to use the code as a name for the program. (But just in this context.) This means $c(P) = P$.

Uncomputability

About functions:

Older view of functions:

A function is presented as a rule for computing.

Ex: $f(x) = 2\sin(x) + 3$

Modern view of functions: A function is a set of pairs $\{(x, y)\}$ such that if (x, y_1) and (x, y_2) are pairs in the function, then $y_1 = y_2$.

Functions can be *uncomputable*

What is computable?

Def: f is computable if and only if there is a Turing Machine such that $f(n) = m \Leftrightarrow T(n)$ halts and returns m .

First proof of uncomputability

The set of computable functions is enumerable. The set of all functions are not!

Let us see some more details:

Let f_1, f_2, f_3, \dots be a list of all computable functions. Take the array

$$\begin{pmatrix} f_1(1) & f_1(2) & f_1(3) & \dots \\ f_2(1) & f_2(2) & f_2(3) & \dots \\ f_3(1) & f_3(2) & f_3(3) & \dots \\ \dots & \dots & \dots & \dots \end{pmatrix}$$

We define a function ϕ such that

$$\begin{cases} \phi(n) = f_n(n) + 1 & \text{if } f_n(n) \text{ is defined} \\ \phi(n) = 1 & \text{if } f_n(n) \text{ is undefined} \end{cases}$$

Then ϕ is uncomputable. (What happens if $\phi = f_k$ for some k ?)

A decision problem is *decidable* if there is some algorithm that decides the problem (correctly) in finite time for every instance.

The opposite is when there, for some reason, is no such algorithm. Then we say that the problem is *undecidable*.

It is usually the case that there is an algorithm that decides the problem for some, but not all, instances.

If output is not Yes/No we normally speak about *computable* and *uncomputable* problems.

Ex. 1: The Post Correspondence Problem

Given a set of pairs of words $\{(x_i, y_i)\}$.

Is there a sequence of integers a_1, a_2, \dots, a_k such that $x_{a_1}x_{a_2} \cdots x_{a_k} = y_{a_1}y_{a_2} \cdots y_{a_k}$?

Example:

$$\left\{ \underbrace{(abb, bbab)}_1, \underbrace{(a, aa)}_2, \underbrace{(bab, ab)}_3, \underbrace{(baba, aa)}_4, \underbrace{(aba, a)}_5 \right\}$$

has solution $a = [2, 1, 1, 4, 1, 5]$:

$$\underbrace{a}_{x_2} \underbrace{abb}_{x_1} \underbrace{abb}_{x_1} \underbrace{baba}_{x_4} \underbrace{abb}_{x_1} \underbrace{aba}_{x_5} = \underbrace{aa}_{y_2} \underbrace{bbab}_{y_1} \underbrace{bbab}_{y_1} \underbrace{aa}_{y_4} \underbrace{bbab}_{y_1} \underbrace{a}_{y_5}$$

but

$$\{(bb, bab), (a, aa), (bab, ab), (baba, aa), (aba, a)\}$$

has no solution.

Ex. 2: The Halting Problem

Given a program P and input X

Does the program P halt when run with input X ?

It doesn't matter what programming language we use. P could be a Turing Machine.

Ex. 3: Some more applied problems:

Program Verification

Given a program P and a specification S for what the program is supposed to do, does the program in fact do it?

Behavior of programs

Can a given line in a program P be reached for some input?

All these problems are undecidable due to close relation to the Halting Problem.

But certain instances of these problems can, of course, be decided.

Proof of decidability/undecidability

Proof of undecidability:

Direct proof

Give a "direct" logical proof why the problem is undecidable.

Reduction

We reduce from a known undecidable problem to our problem. If the reduction is computable, then our problem must be uncomputable.

Proofs of decidability:

- Give an algorithm that decides the problem and show that it works correctly and runs in finite time.

The Halting Problem is undecidable

Suppose there is an algorithm $H(P, X)$ that decides the Halting Problem. Now consider the following program:

$M(P)$

- (1) **if** $H(P, P) = Yes$
- (2) get into an infinite loop
- (3) **else**
- (4) **return**

What happens when we run $M(M)$?

$M(M)$ halts: Then $H(M, M)$ must return No in order for Return to be reached — impossible.

$M(M)$ does not halt: Then $H(M, M)$ returns Yes and then the program will go into the infinite loop and never halt — impossible.

We reach a contradiction. The conclusion is that $H(P, X)$ cannot decide the Halting Problem correctly.

Example of reduction

Almost all variants of the Halting Problem are undecidable
for instance:

Does the program P halt on all inputs?

We can show that there cannot exist an algorithm $HaltAll(P)$ that decides this problem. Indeed, look at the following reduction:

$H(P, X)$

(1) Construct the program Q :

```
Q(Y)
  if  $X = Y$ 
    P(X)
  else
    Halt
```

(2) **return** $HaltAll(Q)$

If $HaltAll(\cdot)$ worked correctly, then we could decide the Halting Problem — impossible.