Undecibability

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 i 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.

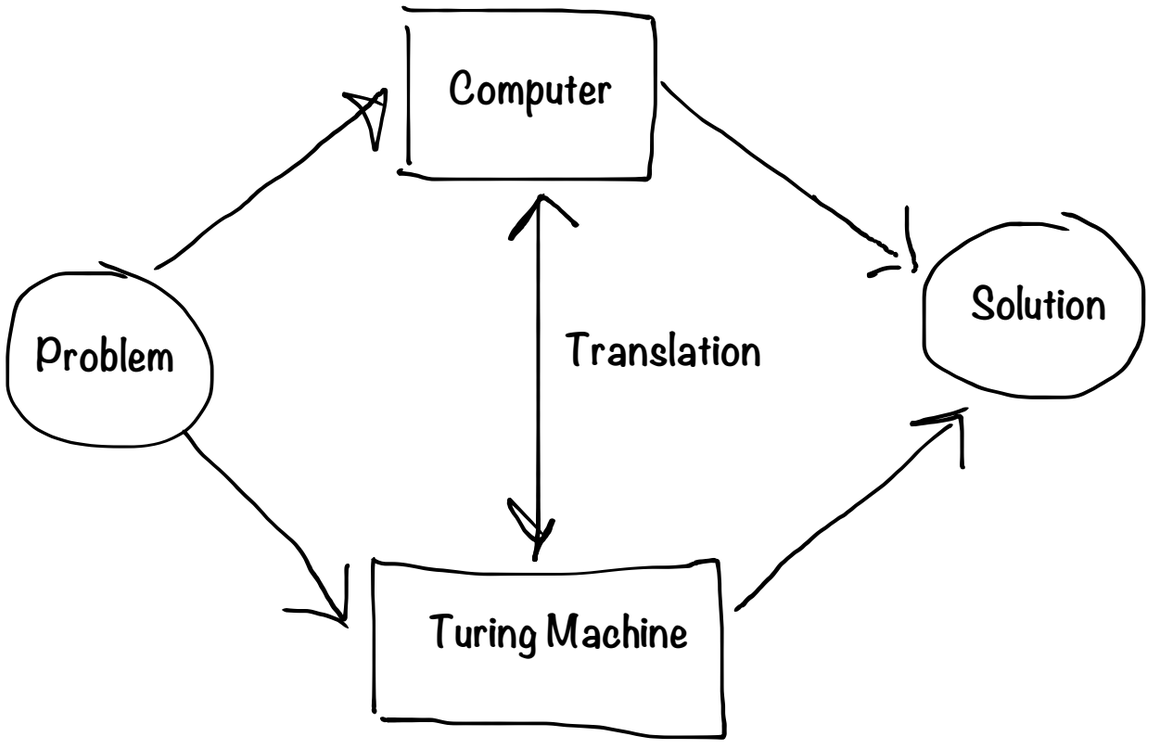
Why Turing Machines?

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

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 $f_1, f_2, f_3, ...$ be a list of all computable functions. Take the array

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

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 $a_1, a_2, \ldots, a_k$ such that $x_{a_1} x_{a_2} \cdots x_{a_k} = y_{a_1} y_{a_2} \cdots y_{a_k}$?

Example:

$$\{\underbrace{(abb, bbab)}_{1}, \underbrace{(a, aa)}_{2}, \underbrace{(bab, ab)}_{3}, \underbrace{(baba, aa)}_{4}, \underbrace{(aba, a)}_{5}\}$$

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.

Proving undecidability

We will use an enumeration $P_1, P_2, ...$ of all possible programs taking an integer as input. We do not know if a program P_n is ''correct'' in some sense. A question we can ask is if P_n will halt if it starts on input m. We can even ask if P_n halts if we start with n as input, that is, if $P_n(n)$ halts.

The diagonal problem
Given an integer n is $P_n(n)$ defined?

This problem is undecidable. Let us assume that there is a program $F(n)$ that decides it. This means that F always halts and returns TRUE if $P_n(n)$ halts and returns FALSE otherwise.

We can then easily define a new function F^* that behaves almost as F:

$$F^*(n) = \begin{cases} TRUE \text{ if } P_n(n) \text{ and returns } TRUE \\ FALSE \text{ otherwise} \end{cases}$$

But it is easy to construct another program $G(n)$ that returns the opposite answer
$F^*(n) = TRUE \Leftrightarrow G(n) = FALSE$
$F^*(n) = FALSE \Leftrightarrow G(n) = TRUE$

This program G must also be in the list. Let us assume that it has number k. What happens when we try to find $G(k)$? We have two possibilities:

1. $G(k) = TRUE$. Then $G(k)$ halts, that is, $P_k(k)$ halts and return TRUE. Then we must have $F^*(k) = TRUE$. But that is impossible by the definition of G.

2. $G(k) = FALSE$. Then $F^*(k) = TRUE$. This means that $P_k(k)$ halts and returns TRUE. But this is impossible since $P_k(k) = G(k)$.

The conclusion is that F can not work correctly, or stated in another way, F does not exist!

More undecidability

Special halting problem

Input is a program P_n and an integer m. Does $P_n(m)$ halt?

It can be seen that this problem is undecidable. Assume that there is a program $F(n, m)$ that decides the problem. If we run $F(n, n)$ we would get a solution to the diagonal problem, which is impossible.

General halting problem

In this problem we take as input the code $c(P)$ of a program P and an input string x. We want to decide if $P(x)$ halts or not.

We can see that we can reduce the special halting problem to this problem. Given n and m we can find the code $c(P_n)$ and write m as a string and give this input to the general halting problem.

We now give another proof of the undecidability of the general halting problem:

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.

Another reduction

Does the program P halt on empty input?

We can again reduce the halting problem to this problem. Assume $HaltEmpty(P)$ decides this problem.

H(P, X)
(1) Create the program Q:
(2) $Q(Y)$
(3) Regardless of input do $P(X)$
(4) **return** $HaltEmpty(Q)$

We see again that we can decide the halting problem, which is impossible.

Semantically equivalent programs

We say that two programs P_1 and P_2 are semantically equivalent if they return the same output (including not halting) on all input. Can we decide if two programs are semantically equivalent? We cannot since it is possible to reduce Halt on all inputs to this problem. Assume $F(P_1, P_2)$ decides the equivalence problem:

$HaltAll(P)$
(1) Create the program P_1
(2) Run P but skip output
(3) **return** 1
(4) Create the program P_2:
(5) **return** 1
(6) **return** $F(P_1, P_2)$

There is a famous theorem called Rice's theorem that roughly says that any nontrivial semantic property of programs is undecidable.

First order logic is undecidable

Assume that we have a formula ϕ in first order logic. We want to decide if it is logically valid or not. This problem is undecidable. It can be shown that the previous problems can all be reduced to this problem. For instance, if we have a program P and want to know if it stops on all inputs we can construct a formula ψ such that the program stops on all inputs if and only if ψ is logically valid.

Recursive enumerability

Even if the problems we have studied are not decidable, they have another property. They are *recursively enumerable*.

A problem is recursively enumerable if there is an algorithm $F(X)$ such that if X is an input to the problem we have:

If X is a yes-instance, then $F(X)$ halts and returns yes.

If X is a no-instance, then $F(X)$ never halts.

(The problem is that if we run the algorithm we don't not know if it will ever stop.)

Note: Decidable problems are sometimes called *recursive* problems.