Topic: Why NP Problem In polynomial time is too complex to be solved?

Subtopic: 01

Which theorems are extracted from Turing's result by Gödel?

Mathematical proofs had become increasingly powerful by the late nineteenth century. Many of these methods included considering sets with infinite members, and proofs often succeeded because such sets might be regarded as entire existing wholes.

The book by Roger states that Hilbert's idea was to compile a list of axioms and procedural rules that encompassed all types of sound mathematical reasoning. He wished to be able to prove either P or ~P depending upon whether P is true or false.

Gödel disproved this idea and He put forth two incompleteness theorems to counter Hilbert's ideas. These theorems are related to limits of provability in formal axiomatic theories. The theorems are widely interpreted as showing that Hilbert's program to find a complete and consistent set of axioms for all mathematics is impossible. The first theorem states that no consistent system of axioms whose theorems can be listed by an effective procedure is capable of proving all truths about the arithmetic of natural numbers. For any such consistent formal system, there will always be statements about natural numbers that are true, but unprovable within the system which means every strong enough formal system is incomplete because there will always be at least one formula that can neither be proved nor disproved. The second theorem which is an extension of the first, shows that the system cannot demonstrate its own consistency which means that if such a formal system admits a proof of its own consistency, then it is actually inconsistent.

Alan Turing reformulated Gödel’s result in terms of algorithms executed by an idealized computer that can read or write one bit at a time. He showed that there are some algorithms that are undecidable by such a Turing machine, that is, it’s impossible to tell whether the machine could complete the calculations in a finite amount of time. And there is no general test to see whether any particular algorithm is undecidable. Similar restrictions apply to real computers too.

The relationship between the Gödel incompleteness theorem and Turing Machines is that if the halting problem which is the problem of determining, from a description of an arbitrary computer program and an input that whether the program will finish running, or continue to run forever is undecidable, then Peano Arithmetic that is the list of axioms of natural numbers is not complete since otherwise, you could solve the halting problem by searching for proofs in PA. The same argument works for any sound computably axiomatizable theory because if the halting problem is not solvable, there must be sentences of this form that are not settled by the theory.

Subtopic: 02

How do you identify that a particular problem is non-recursive?

For a non-recursive set, there is no general algorithmic way of deciding whether or not an element belongs to the set. There are many problems in mathematics that are considered non-recursive. That is why we might have a class of problems to which the answer in each case is either 'yes' or 'no', but for which no general algorithm exists for deciding which of these two is actually the case.

Let us consider the example of a problem that is “The Word Problem” Roger in his book “The Emperor’s New Mind” describes the problem as, suppose that we have some alphabet of symbols, and we consider various strings of these symbols, referred to as words. The words need not in themselves have any meaning, but we shall be given a certain (finite) list of 'equalities' between them which we are allowed to use in order to derive further such 'equalities'. This is done by making substitutions of words from the initial list into other (normally longer) words which contain them as portions. Each such portion may be replaced by another portion that is deemed to be equal to it according to the list. The problem is then to decide, for some given pair of words, whether or not they are 'equal' according to these rules.

Let us take the same example that Roger took in his book, then our initial list is

EAT = AT ATE=A LATER = LOW

PAN = PILLOW CARP = ME

And from these initial lists, we can extract LAP = LEAP by doing multiple substitutions. But can we go from one word to another if we are given some words, i.e. can we go from carpet to meat? Or from Caterpillar to man... In the case of the caterpillar to man, we can but we cannot go from carpet to meat.

To prove that we can go from one word to another we only need to exhibit a string of equalities where each word is obtained from the preceding one by use of an allowed relation. And this is it, we have a clear algorithm to prove if two words are equal but what if we need to prove that we cannot go from a particular word to some other word? That’s somewhat tricky and one has to resort to arguments about the rules that are given. One way to prove two words unequal is to check if their sum of numbers of As, Ws, and Ms is equal. Because in every 'equality' in our initial list, the number of As plus the number of Ws plus the number of Ms should be the same on each side. Thus the total number of As, Was, and Ms cannot change throughout any succession of allowed substitutions. But there is no such obvious algorithm in general that can prove if words are not equal, we have to resort to intelligence to reach a conclusion. Thus it turns out that there is no single algorithm that can be used universally for all possible choices of the initial list. In this sense there is no algorithmic solution to the word problem and that it also belongs to non-recursive mathematics.

Subtopic: 03

Why is there no algorithm to solve NP problems in polynomial time? Would we ever be able to find such an algorithm?

Mathematical problems that are algorithmic in nature also have some classes of problems that are more difficult to solve algorithmically than others. This is what complexity theory is all about. It is concerned with infinite families of problems where there would be a general algorithm for finding answers to all the problems of one single family.

Steps of computation depend on the type of computing machine on which any sort of algorithm runs. This can cause uncertainties which is why experts have made a categorization of the possible ways that N, which is the greatest number of steps that the algorithm takes, can increase. And one such category of N to increase is P which is called polynomial time. Now those algorithms that are not in P are referred to as NP. The N in NP refers to nondeterministic Turing machines and there are various types of NPs. A problem is said to be NP-hard if everything in NP can be transformed in polynomial time into it even though it may not be in NP.

One of the examples of NP problems is the Hamiltonian circuit. A Hamiltonian circuit is simply a closed-up route (or loop) consisting solely of edges of the graph, and which passes exactly once through each vertex. The Hamiltonian circuit problem is to decide, for any given graph, whether or not a Hamiltonian circuit exists, and to display one explicitly whenever one does exist. Another example is the traveling salesman problem, where a polynomial-time solution of the traveling salesman problem would lead to polynomial-time solutions to all other NP problems.

If there was some algorithm that could solve NP problems in polynomial time we could compute any NP problem in polynomial time which would have meant that P = NP.
But till now it has been impossible to have such an algorithm. What we're trying to say here is that P!=NP implies that there are some problems that CAN be verified in polynomial time but at the moment cannot be solved in polynomial time, and this is one of the unsolved problems in computer science today.

Subtopic: 04

Why NP Problem in polynomial time is too complex to be solved?

Everything that we talked about the nature, existence, and limitations of algorithms in the previous videos was on a principle level, but now we need to understand whether these algorithms even exist practically or not. Even for problems where it is clear that algorithms exist and how such algorithms can be constructed, it may require much ingenuity and hard work to develop them into something usable.

NP problems have not been solved in polynomial time till now. We have found ways to approximate a solution or randomly select to at least find an answer close to being solved in polynomial time. But the question is, will this problem ever be solved?
Why is it so complex and why has no one still been able to solve it?

The answer to this can be explained like this. Let’s consider a deterministic machine that begins with a start state, does some configuration in sequential form and has an accept state, let's compare this to a straight line. Now when we talk about a non-deterministic machine (which we need to keep in mind cannot exist in real life), it has many solutions and fans out, just like the branches of this tree or this Chinese fan, and tries out all the solutions. Since this is not a realistic model, we will need a sequential machine in order to simulate it. However, this might pose a problem as the number of branches it fans out to can be infinite, meaning the sequential machine will have to run infinitely, making it impossible in reality. So the question basically here is if determinism and non-determinism are equal and intuitively speaking, it seems they are not.

Well this problem might get solved someday in future. Because, with the passing of every decade, humans tend to understand computation better. If we keep moving forward in this direction and improve our understanding of computation, then who knows one day we'll be able to take the step from using approximations to solve these problems to actually solving them.