Decidable and Undecidable Problems

Angus Macintyre
QMUL, London

May 2011

Turing

I chose to speak about **undecidable problems** because we are within a year of the centenary of Turing's birth.

He is of basic importance in the subject because he gave the first precise definition of idealized machines (Turing machines) allowing an exact definition of the **decidability** of mathematical problems, and thereby opening the possibility of proving that certain problems are **undecidable**.

A useful, though not quite exact, analogy is with the effort that went into having a sufficiently precise definition of

- Number constructible using ruler and compasses
- Equation solvable by radicals

Impossibility Theorems

Number constructible using ruler and compasses: Greek geometers looked for constructions that would trisect an arbitrary angle, and for constructions that would take a circle and construct a square with the same area.
It took until 1837 before one could prove there was no ruler and compass construction for the trisection problem, and around 1882 before one knew the circle could not be squared by ruler and compass or indeed by a huge variety of other constructions.

Equation solvable by radicals: it took until 1824 till Abel showed that there was no formula in terms of radicals that gave the solution of the general polynomial equation of the fifth degree.
For quadratics, the solution was known in antiquity, and for third and fourth degree equations was discovered by Italians in the 16th century.

The most basic decidable problems. Addition

The most basic operation in common experience is addition of integers written in decimal notation.

What is involved? One needs to know the addition tables for numbers no greater than 9, and the remainders after division by 10 of numbers no more than 19.
Then one simply writes one decimal under the other with their last digits lined up, thus:

$$\begin{array}{r} 237862 \\ 140870 \end{array}$$

Now one simply does the standard procedure of
(i) adding coordinatewise from right
(ii) writing the result if less than 10, otherwise taking remainder modulo 10,
(iii) writing the remainder and then carrying 1 to next place,
(iv) repeat in the next position

Addition - cont

(i) adding coordinatewise from right
(ii) writing the result if less than 10, otherwise taking remainder modulo 10,
(iii) writing the remainder and then carrying 1 to next place,
(iv) repeat in the next position

$$\begin{array}{r} {\scriptstyle (1\ 1)} \\ 2\,3\,7\,8\,6\,2 \\ 1\,4\,0_1 8_1 7\,0 \\ \hline 3\,7\,8\,7\,3\,2 \end{array}$$

The process terminates in finitely many steps, needs no ingenuity,and is uniformly defined for all inputs. We recognize it as one of the most basic **algorithms**, or **decision procedures**.

Other familiar algorithms

There is a similar, but more complex procedure for **multiplication**.

To **exponentiate**, say to compute m^n, one can use repeated multiplication. This naive process terminates,and can be described by a "programme" quite independent of inputs. It needs no ingenuity,and proceeds deterministically.

For **long division with remainder**, one has the programme "Euclidean Algorithm" going back to the Greeks (and one could have a more brutal algorithm based just on enumerating cases).

Here one touches on the basic fact that an algorithm for a specific problem is in no way unique, and some may be much more rapid than others, or require less storage, etc.
Such issues were not treated in the theoretical literature in the 1930's, but dominate the subject now.

Primes and Factorization

Let n be given, bigger than 1.

We know that if n has any proper factorization $n = m_1.m_2$ then the factors are less than n.
So we could simply run through all such pairs of integers less than n , multiply them, and check to see if the result is n.
If this works for none, then n is prime, and we are done.
Otherwise, take first factorization that comes up, and now try to factor the factors.
And so on..

This will yield, very slowly, the **factorization of** n.
No ingenuity is required, the process stops and succeeds in giving the right answer. And the process has a finite description.

Here again we have an algorithm, and a decidable problem. But one may well wonder now how slow it is, or how slow any successful algorithm is.....

The Exact Definition of a Turing machine

The basic idea is of astounding simplicity. The idealized component is a **two-way infinite tape, marked into squares**. An actual machine M has a scanner that looks at exactly one square of the tape at a time. M operates on strings from a **finite** alphabet A of symbols. It has a finite number of **states**, and a finite **programme** of instructions of the form:
if in state q, and scanning a square with symbol b on it, replace b by symbol c, move the tape one square to RIGHT (respectively LEFT) and go into state r.

There will be some special states, e.g a BEGIN state and an END state (and when the machine reaches the END state it stops).

On the tape at any given time there will be only finitely many symbols, and all other squares will be construed as occupied by a special nullsymbol.The instructions can include the case when b or c is the nullsymbol.

Computable Functions

We now construe M as **calculating a function of finite strings** as follows.

- Put string σ on the tape,and have the machine scan the square with the leading symbol of σ on it.
- Put machine in BEGIN state. It will start running according to the list of instructions, and will either continue running indefinitely, or reach and END state, in which case it stops.
- In the first case, the function is undefined on the input.
  In the second case, the function is undefined if the scanned square is blank, and otherwise the value of the function is the longest string, reading to right from scanned square, where no blanks occur.

If you want to compute functions of two variables instead, you put σ followed by blank, followed by τ on the tape, and proceed as above. And so on for functions of k variables.

Computable Functions, cont

Now a k-ary function from strings to strings is **Turing computable** if it is defined by a machine as above.

In the case of integers, you may use the alphabet (1,2, ... 9) and identify numbers with strings, to get a notion of computable functions on the integers .

Fact: You get the same class of functions for any other sensible identification of numbers with strings (e.g 2-adic notation).

Alonzo Church, around the same time, gave another definition of computable in terms of lambda calculus, a notion nowadays much more relevant to computer science than Turing's. But the main thing is that **the two notions give the same class of functions**.
Moreover, there is widespread agreement that this is the "correct" definition of computable function.

Church-Turing Thesis

This is the so-called **Church-Turing Thesis**. No one has ever come up with an algorithm for computing a function that does not fall under this classification.

Recently, however, new ideas relating to physical computation have merged, and the picture is murkier.

For this talk, *computable* is *Turing computable*. Moreover, a set X of strings is computable if the function which gives 1 on the set, and 0 off the set, is computable, where 0,1 are symbols not in A.

A quotation by Emil Post

Actually the work done by Church and others carries this identification considerably beyond the working hypothesis stage.

But to mask this identification under a definition hides the fact that a fundamental discovery in the limitations of the mathematicizing power of Homo Sapiens has been made and blinds us to the need of its continual verification.

The Earlier Examples

It is a simple exercise to show that all the sets and functions discussed earlier are computable, as are

- factorial
- the nth prime,
- the first prime after n,

and so on.

More involved decidable examples

It is easy to extend this to get a good definition of computable on the set $\mathbb{Z}$ of all integers, positive, negative or zero.

Now consider the set of all systems of linear equations in k variables, with coefficients in $\mathbb{Z}$, which have nontrivial solutions in $\mathbb{Z}$. It needs a certain amount of linear algebra and elementary number theory to show that this set is computable.

Another example is the set of equations of the form $x^2 - dy^2 = 1$, for d a positive integer. *For which d is this solvable in integers?* This is again computable, because of a nontrivial theorem (concerning the erroneously called Pell's equation).

More decidable problems

- $x^2 - dy^2 = 1$
  It is noteworthy that for these problems, the least solution may be very large.
  For example, when d $= 61$, the least solution is (1766319049, 226153980). In fact, for all square-free d there is a solution, which is why the problem is rapidly decidable, but lots of number theory underlies this, and one does not need the delicate information on the smallest solution.
- Even harder is the problem for $y^2 = x^3 + k$. Alan Baker (Fields Medallist 1966) showed that the k for which this is solvable is decidable.

A Major Undecidability Result

The problem had been posed in 1900 by Hilbert (his 10th Problem) before the precise definition was known.

Is there an algorithm to decide if a system of equations with integer coefficients has an integer solution?

In 1970, the 23 year old Yuri Matejasevic showed that there is no such algorithm (assuming Church-Turing Thesis).

An amazing spinoff

The set of positive values taken by the polynomial

$$(k+2)\{1-[wz+h+jq]^2-[(gk+2g+k+1)(h+j)+hz]^2-$$
$$[2n+p+q+ze]^2-[16(k+1)^3(k+2)(n+1)^2+1f^2]^2-$$
$$[e^3(e+2)(a+1)^2+1-o^2]^2-[(a^21)y^2+1x^2]^2-$$
$$[16r^2y^4(a^2-1)+1-u^2]^2-$$
$$[((a+u^2(u^2-a))^2-1)(n+4dy)^2+1-(x+cu)^2]^2-$$
$$[n+l+v-y]^2-[(a^2-1)l^2+1-m^2]^2-[ai+k+1-l-i]^2-$$
$$[p+l(a-n-1)+b(2an+2a-n^2-2n-2)-m]^2-$$
$$[q+y(a-p-1)+s(2ap+2a-p^2-2p-2)-x]^2-$$
$$[z+pl(a-p)+t(2ap-p^2-1)-pm]^2\}$$

is exactly the set of primes!!!

No mention of computability, but the proof goes through considerations of computability. The result was unknown to number theorists!

A Major Decidability Result

In contrast to the earlier results about integers, this is a result about geometry. It was proved by Alfred Tarski in the 1930's but not published till 1948 due to the devastation of the war.

It says that **there is an algorithm for testing the solvability in the real numbers of systems of polynomial equations with integer coefficients**. Using cartesian coordinates one can translate into this form most questions in Euclidean geometry.

The proof, and not simply the result, has had many applications to the study of real geometry, differential equations, etc..

Something I did

Tarski has asked if one could extend his result to take account of the **exponential function** on the reals.

This remained open for some 60 years, till Wilkie and I showed in 1992 that the answer is yes, assuming a famous conjecture in transcendental number theory (generalizing the result that the circle cannot be squared).
This allows one to decide certain questions in hyperbolic geometry that cannot be obtained via Tarski's algorithm.

The proofs in this area have had spectacular applications in Lie theory and number theory, and also in neural networks.

What Turing did

Turing showed unsolvability of the Halting problem. You want to print in to some machine the code for the programme of another, and ask whether the coded machine halts on input 1.

Using ideas of Godel, he showed that there is no machine that gives the right answer.

He worked postwar on classical decision problems, notably the Word Problem for Groups, formulated by Dehn in 1910 in connection with problems in topology. Though he obtained very good results for cancellative semigroups, he could not crack the original problem. It was solved in 1955 by Peter Novikov (year after Turing died).

He also gave some positive algorithms for Lie groups using serious representation theory.

Concerns About length of Computation

One of the most fundamental features of Turing computability is that **computable functions may be very costly to compute** in terms of time, or space resource. A natural measure of complexity for a machine M computing a function f (say of one variable) is the function F, where F(n) is the smallest m so that for all σ of length no more than n M computes $f(\sigma)$ in no more than m steps. If F(n) is bounded by a polynomial in n the we say M computes f in polynomial time, and then that f is polynomial time computable. The function 2^n is computable, but not polynomial time computable.

A function which is not polynomial time computable is not in general realistically computable (and indeed if the polynomial degree is large even a polynomial time computabe function may be hard to compute).

Primality Testing

It was a major open problem for many years **whether or not the set of primes is polynomial time computable**, and it was a sensation when in 2002 this result was proved (using serious but not exceptionally hard number theory) by Agrawal, Kayal and Saxena.

Whether factorization can be achieved in polynomial time remains an open problem, and it is of major importance for security that the problem not be doable in polynomial time.

Deterministic and NonDeterministic Computation

The computational model sketched earlier is deterministic, in the sense that if there is a next action of the machine, it is uniquely determined.

One may naturally relax this, by allowing, in a definite way, finitely many next moves. (I omit the details). In this way one gets the notion of a nondeterministic Turing machine.
What is relevant now is the set of inputs from which the machine reaches HALT (this could happen, for a given input, in several ways). Thus we get the notions of nondeterministically computable sets and functions.

It turns out, for Turing computation, that these notions are not more general than the deterministic versions. This is proved by coding devices à la Gödel.

P and NP

However, Gödel spotted, in the 1950's, an intriguing problem, which he formulated in a letter to the dying von Neumann.
Is the notion of **polynomial time nondeterministic computability** more general than that of **polynomial time deterministic computatibility**?

This has resisted attack, and is now a Million Problem of the Clay Foundation. It is generally believed that the answer is YES .
A NO answer would be unpalatable for security.....

Other Meanings of Decidable

The discovery that made Gödel world-famous is related to Turing's discoveries, and precedes them by a few years.
It involves the discovery of **natural mathematical statements which cannot be settled on the basis of the axiom systems** currently accepted by most mathematicians.

His most general theorem is about **computable** systems of axioms for arithmetic or set theory. Later work by him, Paul Cohen and now hundreds of others exhibit specific statements, previously considered by set theorists, which cannot be settled on the basis of the present axioms.
These have little if anything to do with **computability**, and have engendered a large (and largely unimpressive) philosophical literature.

For reference

Turing, A.M. (1936),
On Computable Numbers, with an Application to the Entscheidungsproblem,
Proceedings of the London Mathematical Society, 2 42 (1): 230-65, 1937.