Contents

Chapter 1

Post correspondence problem

Not to be confused with the other Post's problem on the existence of incomparable r.e. Turing degrees.
Not to be confused with PCP theorem.

The **Post correspondence problem** is an undecidable decision problem that was introduced by Emil Post in 1946.[1] Because it is simpler than the halting problem and the *Entscheidungsproblem* it is often used in proofs of undecidability.

1.1 Definition of the problem

The input of the problem consists of two finite lists $\alpha_1, \ldots, \alpha_N$ and $\beta_1, \ldots, \beta_N$ of words over some alphabet A having at least two symbols. A solution to this problem is a sequence of indices $(i_k)_{1 \le k \le K}$ with $K \ge 1$ and $1 \le i_k \le N$ for all k , such that

$$\alpha_{i_1} \ldots \alpha_{i_K} = \beta_{i_1} \ldots \beta_{i_K}.$$

The decision problem then is to decide whether such a solution exists or not.

1.1.1 Alternative definition

In the above definition, a solution can be seen as a nonempty word on the alphabet $\{1, \ldots, N\}$, and each of the two given lists of words on alphabet A can be seen as defining a homomorphism that maps words on $\{1, \ldots, N\}$ to words on A :

$$g : (i_1, \ldots, i_K) \mapsto \alpha_{i_1} \ldots \alpha_{i_K}$$

$$h : (i_1, \ldots, i_K) \mapsto \beta_{i_1} \ldots \beta_{i_K}.$$

This gives rise to an equivalent alternative definition often found in the literature, according to which any two homomorphisms g, h with a common domain and a common codomain form an instance of the Post correspondence problem, which now asks whether there exists a nonempty word w in the domain such that

$$g(w) = h(w)$$

1.2 Example instances of the problem

1.2.1 Example 1

Consider the following two lists:

A solution to this problem would be the sequence (3, 2, 3, 1), because

$$\alpha_3\alpha_2\alpha_3\alpha_1 = bba{+}ab{+}bba{+}a = bbaabbbaa = bb{+}aa{+}bb{+}baa = \beta_3\beta_2$$

Furthermore, since (3, 2, 3, 1) is a solution, so are all of its "repetitions", such as (3, 2, 3, 1, 3, 2, 3, 1), etc.; that is, when a solution exists, there are infinitely many solutions of this repetitive kind.

However, if the two lists had consisted of only α_2, α_3 and β_2, β_3 from those sets, then there would have been no solution (the last letter of any such α string is not the same as the letter before it, whereas β only constructs pairs of the same letter).

A convenient way to view an instance of a Post correspondence problem is as a collection of blocks of the form

there being an unlimited supply of each type of block. Thus the above example is viewed as

where the solver has an endless supply of each of these three block types. A solution corresponds to some way of laying blocks next to each other so that the string in the top cells corresponds to the string in the bottom cells. Then the solution to the above example corresponds to:

1.2.2 Example 2

Again using blocks to represent an instance of the problem, the following is an example that has infinitely many solutions in addition to the kind obtained by merely "repeating" a solution.

In this instance, every sequence of the form (1, 2, 2, . . ., 2, 3) is a solution (in addition to all their repetitions):

1.3 Proof sketch of undecidability

The most common proof for the undecidability of PCP describes an instance of PCP that can simulate the computation of a Turing machine on a particular input. A match will occur if and only if the input would be accepted by the Turing machine. Because deciding if a Turing machine will accept an input is a basic undecidable problem, PCP cannot be decidable either. The following discussion is based on Michael Sipser's textbook *Introduction to the Theory of Computation*.[2]

In more detail, the idea is that the string along the top and bottom will be a computation history of the Turing machine's computation. This means it will list a string describing the initial state, followed by a string describing the next state, and so on until it ends with a string describing an accepting state. The state strings are separated by some separator symbol (usually written #). According to the definition of a Turing machine, the full state of the machine consists of three parts:

- The current contents of the tape.
- The current state of the finite state machine which operates the tape head.
- The current position of the tape head on the tape.

Although the tape has infinitely many cells, only some finite prefix of these will be non-blank. We write these down as part of our state. To describe the state of the finite control, we create new symbols, labelled q_1 through qk, for each of the finite state machine's k states. We insert the correct symbol into the string describing the tape's contents at the position of the tape head, thereby indicating both the tape head's position and the current state of the finite control. For the alphabet {0,1}, a typical state might look something like:

$101101110q_7 00110$.

A simple computation history would then look something like this:

$q_0 101\#1q_4 01\#11q_2 1\#1q_8 10$.

We start out with this block, where x is the input string and q_0 is the start state:

The top starts out "lagging" the bottom by one state, and keeps this lag until the very end stage. Next, for each symbol a in the tape alphabet, as well as #, we have a "copy" block, which copies it unmodified from one state to the next:

We also have a block for each position transition the machine can make, showing how the tape head moves, how the finite state changes, and what happens to the surrounding symbols. For example, here the tape head is over a 0 in state 4, and then writes a 1 and moves right, changing to state 7:

Finally, when the top reaches an accepting state, the bottom needs a chance to finally catch up to complete the match. To allow this, we extend the computation so that once an accepting state is reached, each subsequent machine step will cause a symbol near the tape head to vanish, one at a time, until none remain. If qf is an accepting state, we can represent this with the following transition blocks, where a is a tape alphabet symbol:

There are a number of details to work out, such as dealing with boundaries between states, making sure that our initial tile goes first in the match, and so on, but this shows the general idea of how a static tile puzzle can simulate a Turing machine computation.

The previous example

$q_0 101\#1q_4 01\#11q_2 1\#1q_8 10$.

is represented as the following solution to the Post correspondence problem:

1.4 Variants

Many variants of PCP have been considered. One reason is that, when one tries to prove undecidability of some new problem by reducing from PCP, it often happens that the first reduction one finds is not from PCP itself but from an apparently weaker version.

- The problem may be phrased in terms of monoid morphisms f, g from the free monoid B^* to the free monoid A^* where B is of size n. The problem is to determine whether there is a word w in B^+ such that $f(w) = g(w)$.[3]
- The condition that the alphabet A have at least two symbols is required since the problem is decidable if A has only one symbol.
- A simple variant is to fix n, the number of tiles. This problem is decidable if $n \leq 2$, but remains undecidable for $n \geq 5$. It is unknown whether the problem is decidable for $3 \leq n \leq 4$.[4]
- The ***circular*** **Post correspondence problem** asks whether indexes $i_1, i_2, \ldots$ can be found such that $\alpha_{i_1} \cdots \alpha_{i_k}$ and $\beta_{i_1} \cdots \beta_{i_k}$ are conjugate words, i.e., they are equal modulo rotation. This variant is undecidable.[5]
- One of the most important variants of PCP is the ***bounded*** **Post correspondence problem**, which asks if we can find a match using no more than k tiles, including repeated tiles. A brute force search solves the problem in time $O(2^k)$, but this may be

difficult to improve upon, since the problem is NP-complete.[6] Unlike some NP-complete problems like the boolean satisfiability problem, a small variation of the bounded problem was also shown to be complete for RNP, which means that it remains hard even if the inputs are chosen at random (it is hard on average over uniformly distributed inputs).[7]

- Another variant of PCP is called the ***marked* Post Correspondence Problem**, in which each *ui* must begin with a different symbol, and each *vi* must also begin with a different symbol. Halava, Hirvensalo, and de Wolf showed that this variation is decidable in exponential time. Moreover, they showed that if this requirement is slightly loosened so that only one of the first two characters need to differ (the so-called 2-marked Post Correspondence Problem), the problem becomes undecidable again.[8]

- The **Post Embedding Problem** is another variant where one looks for indexes $i_1, i_2, \ldots$ such that $\alpha_{i_1} \cdots \alpha_{i_k}$ is a (scattered) subword of $\beta_{i_1} \cdots \beta_{i_k}$. This variant is easily decidable since, when some solutions exist, in particular a length-one solution exists. More interesting is the **Regular** Post Embedding Problem, a further variant where one looks for solutions that belong to a given regular language (submitted, e.g., under the form of a regular expression on the set $\{1, \ldots, N\}$). The Regular Post Embedding Problem is still decidable but, because of the added regular constraint, it has a very high complexity that dominates every multiply recursive function.[9]

- The **Identity Correspondence Problem** (ICP) asks whether a finite set of pairs of words (over a group alphabet) can generate an identity pair by a sequence of concatenations. The problem is undecidable and equivalent to the following Group Problem: is the semigroup generated by a finite set of pairs of words (over a group alphabet) a group.[10]

1.5 References

[1] E. L. Post (1946). "A variant of a recursively unsolvable problem" (PDF). *Bull. Amer. Math. Soc* **52**.

[2] Michael Sipser (2005). "A Simple Undecidable Problem". *Introduction to the Theory of Computation* (2nd ed.). Thomson Course Technology. pp. 199–205. ISBN 0-534-95097-3.

[3] Salomaa, Arto (1981). *Jewels of Formal Language Theory*. Pitman Publishing. pp. 74–75. ISBN 0-273-08522-0. Zbl 0487.68064.

[4] T. Neary (2015). "Undecidability in Binary Tag Systems and the Post Correspondence Problem for Five Pairs of Words". In Ernst W. Mayr and Nicolas Ollinger. *32nd International Symposium on Theoretical Aspects of Computer Science (STACS 2015)*. STACS 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. pp. 649–661. doi:10.4230/LIPIcs.STACS.2015.649.

[5] K. Ruohonen (1983). "On some variants of Post's correspondence problem". *Acta Informatica* (Springer) **19** (4): 357–367. doi:10.1007/BF00290732.

[6] Michael R. Garey; David S. Johnson (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman. p. 228. ISBN 0-7167-1045-5.

[7] Y. Gurevich (1991). "Average case completeness". *J. Comp. Sys. Sci.* (Elsevier Science) **42** (3): 346–398. doi:10.1016/0022-0000(91)90007-R.

[8] V. Halava; M. Hirvensalo; R. de Wolf (2001). "Marked PCP is decidable". *Theor. Comp. Sci.* (Elsevier Science) **255**: 193–204. doi:10.1016/S0304-3975(99)00163-2.

[9] P. Chambart; Ph. Schnoebelen (2007). "Post embedding problem is not primitive recursive, with applications to channel systems". *Lecture Notes in Computer Science*. Lecture Notes in Computer Science (Springer) **4855**: 265–276. doi:10.1007/978-3-540-77050-3_22. ISBN 978-3-540-77049-7.

[10] Paul C. Bell; Igor Potapov (2010). "On the Undecidability of the Identity Correspondence Problem and its Applications for Word and Matrix Semigroups". *International Journal of Foundations of Computer Science* (World Scientific) **21.6**: 963–978. arXiv:0902.1975. doi:10.1142/S0129054110007660.

1.6 External links

- Eitan M. Gurari. *An Introduction to the Theory of Computation*, Chapter 4, Post's Correspondence Problem. A proof of the undecidability of PCP based on Chomsky type-0 grammars.
- Online PHP Based PCP Solver
- PCP AT HOME
- PCP - a nice problem

Chapter 2

Halting problem

In computability theory, the **halting problem** is the problem of determining, from a description of an arbitrary computer program and an input, whether the program will finish running or continue to run forever.

Alan Turing proved in 1936 that a general algorithm to solve the halting problem for *all* possible program-input pairs cannot exist. A key part of the proof was a mathematical definition of a computer and program, which became known as a Turing machine; the halting problem is *undecidable* over Turing machines. It is one of the first examples of a decision problem.

Jack Copeland (2004) attributes the term *halting problem* to Martin Davis.[1]

2.1 Background

The halting problem is a decision problem about properties of computer programs on a fixed Turing-complete model of computation, i.e., all programs that can be written in some given programming language that is general enough to be equivalent to a Turing machine. The problem is to determine, given a program and an input to the program, whether the program will eventually halt when run with that input. In this abstract framework, there are no resource limitations on the amount of memory or time required for the program's execution; it can take arbitrarily long, and use arbitrarily as much storage space, before halting. The question is simply whether the given program will ever halt on a particular input.

For example, in pseudocode, the program

```
while (true) continue
```

does not halt; rather, it goes on forever in an infinite loop. On the other hand, the program

```
print "Hello, world!"
```

does halt.

While deciding whether these programs halt is simple, more complex programs prove problematic.

One approach to the problem might be to run the program for some number of steps and check if it halts. But if the program does not halt, it is unknown whether the program will eventually halt or run forever.

Turing proved no algorithm exists that always correctly decides whether, for a given arbitrary program and input, the program halts when run with that input; the essence of Turing's proof is that any such algorithm can be made to contradict itself, and therefore cannot be correct.

2.2 Importance and consequences

The halting problem is historically important because it was one of the first problems to be proved undecidable. (Turing's proof went to press in May 1936, whereas Alonzo Church's proof of the undecidability of a problem in the lambda calculus had already been published in April 1936.) Subsequently, many other undecidable problems have been described; the typical method of proving a problem to be undecidable is with the technique of *reduction*. To do this, it is sufficient to show that if a solution to the new problem were found, it could be used to decide an undecidable problem by transforming instances of the undecidable problem into instances of the new problem. Since we already know that *no* method can decide the old problem, no method can decide the new problem either. Often the new problem is reduced to solving the halting problem. (Note: the same technique is used to demonstrate that a problem is NP complete, only in this case, rather than demonstrating that there is no solution, it demonstrates there is no *polynomial time* solution, assuming $P \neq NP$).

For example, one such consequence of the halting problem's undecidability is that there cannot be a general algorithm that decides whether a given statement about natural numbers is true or not. The reason for this is that the proposition stating that a certain program will halt given a certain input can be converted into an equivalent statement about natural numbers. If we had an algorithm that could find the truth value of every statement about natural numbers, it could certainly find the truth value of this one; but that would determine whether the original program halts, which is impossible, since the halting problem is undecidable.

Rice's theorem generalizes the theorem that the halting problem is unsolvable. It states that for *any* non-trivial property, there is no general decision procedure that, for all programs, decides whether the partial function implemented by the input program has that property. (A partial function is a function which may not always produce a result, and so is used to model programs, which can either produce results or fail to halt.) For example, the property "halt for the input 0" is undecidable. Here, "non-trivial" means that the set of partial functions that satisfy the property is neither the empty set nor the set of all partial functions. For example, "halts or fails to halt on input 0" is clearly true of all partial functions, so it is a trivial property, and can be decided by an algorithm that simply reports "true." Also, note that this theorem holds only for properties of the partial function implemented by the program; Rice's Theorem does not apply to properties of the program itself. For example, "halt on input 0 within 100 steps" is *not* a property of the partial function that is implemented by the program—it is a property of the program implementing the partial function and is very much decidable.

Gregory Chaitin has defined a halting probability, represented by the symbol Ω, a type of real number that informally is said to represent the probability that a randomly produced program halts. These numbers have the same Turing degree as the halting problem. It is a normal and transcendental number which can be defined but cannot be completely computed. This means one can prove that there is no algorithm which produces the digits of Ω, although its first few digits can be calculated in simple cases.

While Turing's proof shows that there can be no general method or algorithm to determine whether algorithms halt, individual instances of that problem may very well be susceptible to attack. Given a specific algorithm, one can often show that it must halt for any input, and in fact computer scientists often do just that as part of a correctness proof. But each proof has to be developed specifically for the algorithm at hand; there is no *mechanical, general way* to determine whether algorithms on a Turing machine halt. However, there are some heuristics that can be used in an automated fashion to attempt to construct a proof, which succeed frequently on typical programs. This field of research is known as automated termination analysis.

Since the negative answer to the halting problem shows that there are problems that cannot be solved by a Turing machine, the Church–Turing thesis limits what can be accomplished by any machine that implements effective methods. However, not all machines conceivable to human imagination are subject to the Church–Turing thesis (e.g. oracle machines). It is an open question whether there can be actual deterministic physical processes that, in the long run, elude simulation by a Turing machine, and in particular whether any such hypothetical process could usefully be harnessed in the form of a calculating machine (a hypercomputer) that could solve the halting problem for a Turing machine amongst other things. It is also an open question whether any such unknown physical processes are involved in the working of the human brain, and whether humans can solve the halting problem (Copeland 2004, p. 15).

2.3 Representation as a set

The conventional representation of decision problems is the set of objects possessing the property in question. The **halting set**

$K := \{ (i, x) \mid \text{program } i \text{ halts when run on input } x\}$

represents the halting problem.

This set is recursively enumerable, which means there is a computable function that lists all of the pairs (i, x) it contains.[2] However, the complement of this set is not recursively enumerable.[2]

There are many equivalent formulations of the halting problem; any set whose Turing degree equals that of the halting problem is such a formulation. Examples of such sets include:

- $\{ i \mid$ program i eventually halts when run with input $0 \}$
- $\{ i \mid$ there is an input x such that program i eventually halts when run with input $x \}$.

2.4 Sketch of proof

The proof shows there is no total computable function that decides whether an arbitrary program i halts on arbitrary input x; that is, the following function h is not computable (Penrose 1990, p. 57–63):

$$h(i,x) = \begin{cases} 1 & \text{if program } i \text{ input on halts } x, \\ 0 & \text{otherwise.} \end{cases}$$

Here *program* i refers to the i th program in an enumeration of all the programs of a fixed Turing-complete model of computation.

Possible values for a total computable function f arranged in a 2D array. The orange cells are the diagonal. The values of $f(i,i)$ and $g(i)$ are shown at the bottom; U indicates that the function g is undefined for a particular input value.

The proof proceeds by directly establishing that every total computable function with two arguments differs from the required function *h*. To this end, given any total computable binary function *f*, the following partial function *g* is also computable by some program *e*:

$$g(i) = \begin{cases} 0 & \text{if} f(i,i) = 0, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The verification that *g* is computable relies on the following constructs (or their equivalents):

- computable subprograms (the program that computes *f* is a subprogram in program *e*),
- duplication of values (program *e* computes the inputs *i*,*i* for *f* from the input *i* for *g*),
- conditional branching (program *e* selects between two results depending on the value it computes for *f*(*i*,*i*)),
- not producing a defined result (for example, by looping forever),
- returning a value of 0.

The following pseudocode illustrates a straightforward way to compute *g*:

procedure compute_g(i): if f(i,i) == 0 then return 0 else loop forever

Because *g* is partial computable, there must be a program *e* that computes *g*, by the assumption that the model of computation is Turing-complete. This program is one of all the programs on which the halting function *h* is defined. The next step of the proof shows that *h*(*e*,*e*) will not have the same value as *f*(*e*,*e*).

It follows from the definition of *g* that exactly one of the following two cases must hold:

- *f*(*e*,*e*) = 0 and so *g*(*e*) = 0. In this case *h*(*e*,*e*) = 1, because program *e* halts on input *e*.
- *f*(*e*,*e*) ≠ 0 and so *g*(*e*) is undefined. In this case *h*(*e*,*e*) = 0, because program *e* does not halt on input *e*.

In either case, *f* cannot be the same function as *h*. Because *f* was an *arbitrary* total computable function with two arguments, all such functions must differ from *h*.

This proof is analogous to Cantor's diagonal argument. One may visualize a two-dimensional array with one column and one row for each natural number, as indicated in the table above. The value of *f*(*i*,*j*) is placed at column *i*, row *j*. Because *f* is assumed to be a total computable function, any element of the array can be calculated using *f*. The construction of the function *g* can be visualized using the main diagonal of this array. If the array has a 0 at position (*i*,*i*), then *g*(*i*) is 0. Otherwise, *g*(*i*) is undefined. The contradiction comes from the fact that there is some column *e* of the array corresponding to *g* itself. Now assume *f* was the halting function *h*, if *g*(*e*) is defined (*g*(*e*) = 0 in this case), *g*(*e*) halts so *f*(*e*,*e*) = 1. But *g*(*e*) = 0 only when *f*(*e*,*e*) = 0, contradicting *f*(*e*,*e*) = 1. Similarly, if *g*(*e*) is not defined, then halting function *f*(*e*,*e*) = 0, which leads to *g*(*e*) = 0 under *g*'s construction. This contradicts the assumption that *g*(*e*) not being defined. In both cases contradiction arises. Therefore any arbitrary computable function *f* cannot be the halting function *h*.

2.5 Proof as a corollary of the uncomputability of Kolmogorov complexity

The undecidability of the halting problem also follows from the fact that Kolmogorov complexity is not computable. If the halting problem were decidable, it would be possible to construct a program that generated programs of increasing length, running those that halt and comparing their final outputs with a string parameter until one matched (which must happen eventually, as any string can be generated by a program that contains it as data and just lists it); the length of the matching generated program would then be the Kolmogorov complexity of the parameter, as the terminating generated program must be the shortest (or shortest equal) such program.[3]

2.6 Common pitfalls

The difficulty in the halting problem lies in the requirement that the decision procedure must work for all programs and inputs. A particular program either halts on a given input or does not halt. Consider one algorithm that always answers “halts” and another that always answers “doesn't halt”. For any specific program and input, one of these two algorithms answers correctly, even though nobody may know which one.

There are programs (interpreters) that simulate the execution of whatever source code they are given. Such programs can demonstrate that a program does halt if this is the case: the interpreter itself will eventually halt its simulation, which shows that the original program halted. However, an interpreter will not halt if its input program does not halt, so this approach cannot solve the halting problem as stated. It does not successfully answer “doesn't halt” for programs that do not halt.

The halting problem is theoretically decidable for linear bounded automata (LBAs) or deterministic machines with finite memory. A machine with finite memory has a finite number of states, and thus any deterministic program on it must eventually either halt or repeat a previous state:

> *...any finite-state machine, if left completely to itself, will fall eventually into a perfectly periodic repetitive pattern*. The duration of this repeating pattern cannot exceed the number of internal states of the machine... (italics in original, Minsky 1967, p. 24)

Minsky warns us, however, that machines such as computers with e.g., a million small parts, each with two states, will have at least $2^{1,000,000}$ possible states:

> This is a 1 followed by about three hundred thousand zeroes ... Even if such a machine were to operate at the frequencies of cosmic rays, the aeons of galactic evolution would be as nothing compared to the time of a journey through such a cycle (Minsky 1967 p. 25):

Minsky exhorts the reader to be suspicious—although a machine may be finite, and finite automata "have a number of theoretical limitations":

> ...the magnitudes involved should lead one to suspect that theorems and arguments based chiefly on the mere finiteness [of] the state diagram may not carry a great deal of significance. (Minsky p. 25)

It can also be decided automatically whether a nondeterministic machine with finite memory halts on none, some, or all of the possible sequences of nondeterministic decisions, by enumerating states after each possible decision.

2.7 Formalization

In his original proof Turing formalized the concept of *algorithm* by introducing Turing machines. However, the result is in no way specific to them; it applies equally to any other model of computation that is equivalent in its computational power to Turing machines, such as Markov algorithms, Lambda calculus, Post systems, register machines, or tag systems.

What is important is that the formalization allows a straightforward mapping of algorithms to some data type that the algorithm can operate upon. For example, if the formalism lets algorithms define functions over strings (such as Turing machines) then there should be a mapping of these algorithms to strings, and if the formalism lets algorithms define functions over natural numbers (such as computable functions) then there should be a mapping of algorithms to natural numbers. The mapping to strings is usually the most straightforward, but strings over an alphabet with n characters can also be mapped to numbers by interpreting them as numbers in an n-ary numeral system.

2.8 Relationship with Gödel's incompleteness theorems

The concepts raised by Gödel's incompleteness theorems are very similar to those raised by the halting problem, and the proofs are quite similar. In fact, a weaker form of the First Incompleteness Theorem is an easy consequence of the undecidability of the halting problem. This weaker form differs from the standard statement of the incompleteness theorem by asserting that a complete, consistent and sound axiomatization of all statements about natural numbers is unachievable. The "sound" part is the weakening: it means that we require the axiomatic system in question to prove only *true* statements about natural numbers. The more general statement of the incompleteness theorems does not require a soundness assumption of this kind.

The weaker form of the theorem can be proven from the undecidability of the halting problem as follows. Assume that we have a consistent and complete axiomatization of all true first-order logic statements about natural numbers. Then we can build an algorithm that enumerates all these statements. This means that there is an algorithm $N(n)$ that, given a natural number n, computes a true first-order logic statement about natural numbers such that, for all the true statements, there is at least one n such that $N(n)$ yields that statement. Now suppose we want to decide whether the algorithm with representation a halts on input i. By using Kleene's T predicate, we can express the statement "a halts on input i" as a statement $H(a, i)$ in the language of arithmetic. Since the axiomatization is complete it follows that either there is an n such that $N(n) = H(a, i)$ or there is an n' such that $N(n') = \neg H(a, i)$. So if we iterate over all n until we either find $H(a, i)$ or its negation, we will always halt. This means that this gives us an algorithm to decide the halting problem. Since we know that there cannot be such an algorithm, it follows that the assumption that there is a consistent and complete axiomatization of all true first-order logic statements about natural numbers must be false.

2.9 Variants of the halting problem

Many variants of the halting problem can be found in Computability textbooks (e.g., Sipser 2006, Davis 1958, Minsky 1967, Hopcroft and Ullman 1979, Börger 1989). Typically their undecidability follows by reduction from the standard halting problem. However, some of them have a higher degree of unsolvability. The next two examples are typical.

2.9.1 Halting on all inputs

The *universal halting problem*, also known (in recursion theory) as *totality*, is the problem of determining, whether a given computer program will halt *for every input* (the name *totality* comes from the equivalent question of whether the computed function is total). This problem is not only undecidable, as the halting problem, but highly undecidable. In terms of the Arithmetical hierarchy, it is Π_2^0 -complete.[4] This means, in particular, that it cannot be decided even with an oracle for the halting problem.

2.9.2 Recognizing partial solutions

There are many programs that, for some inputs, return a correct answer to the halting problem, while for other inputs they do not return an answer at all. However the problem ″given program *p*, is it a partial halting solver″ (in the sense described) is at least as hard as the halting problem. To see this, assume that there is an algorithm PHSR (″partial halting solver recognizer″) to do that. Then it can be used to solve the halting problem, as follows: To test whether input program *x* halts on *y*, construct a program *p* that on input (*x*,*y*) reports *true* and diverges on all other inputs. Then test *p* with PHSR.

The above argument is a reduction of the halting problem to PHS recognition, and in the same manner, harder problems such as *halting on all inputs* can also be reduced, implying that PHS recognition is not only undecidable, but higher in the Arithmetical hierarchy, specifically Π_2^0 -complete.

2.9.3 Generalized to oracle machines

See also: Turing jump

A machine with an oracle for the halting problem can determine whether particular Turing machines will halt on particular inputs, but they cannot determine, in general, if machines equivalent to themselves will halt.

More generally, there is no oracle machines with oracle to some problem that can determine in general whether a machine with an oracle to the same problem will halt. Thus, for any oracle O, the halting problem for oracle Turing machines with an oracle to O is not O-computable.

2.10 History

Further information: History of algorithms

- 1900: David Hilbert poses his “23 questions” (now known as Hilbert’s problems) at the Second International Congress of Mathematicians in Paris. “Of these, the second was that of proving the consistency of the 'Peano axioms' on which, as he had shown, the rigour of mathematics depended”. (Hodges p. 83, Davis’ commentary in Davis, 1965, p. 108)

- 1920–1921: Emil Post explores the halting problem for tag systems, regarding it as a candidate for unsolvability. (*Absolutely unsolvable problems and relatively undecidable propositions – account of an anticipation*, in Davis, 1965, pp. 340–433.) Its unsolvability was not established until much later, by Marvin Minsky (1967).

- 1928: Hilbert recasts his 'Second Problem' at the Bologna International Congress. (Reid pp. 188–189) Hodges claims he posed three questions: i.e. #1: Was mathematics *complete*? #2: Was mathematics *consistent*? #3: Was mathematics *decidable*? (Hodges p. 91). The third question is known as the *Entscheidungsproblem* (Decision Problem). (Hodges p. 91, Penrose p. 34)

- 1930: Kurt Gödel announces a proof as an answer to the first two of Hilbert’s 1928 questions [cf Reid p. 198]. “At first he [Hilbert] was only angry and frustrated, but then he began to try to deal constructively with the problem... Gödel himself felt—and expressed the thought in his paper—that his work did not contradict Hilbert’s formalistic point of view” (Reid p. 199)

- 1931: Gödel publishes “On Formally Undecidable Propositions of Principia Mathematica and Related Systems I”, (reprinted in Davis, 1965, p. 5ff)

- 19 April 1935: Alonzo Church publishes “An Unsolvable Problem of Elementary Number Theory”, wherein he identifies what it means for a function to be *effectively calculable*. Such a function will have an algorithm, and "...the fact that the algorithm has terminated becomes effectively known ...” (Davis, 1965, p. 100)

- 1936: Church publishes the first proof that the *Entscheidungsproblem* is unsolvable. (*A Note on the Entscheidungsproblem*, reprinted in Davis, 1965, p. 110.)

- 7 October 1936: Emil Post's paper “Finite Combinatory Processes. Formulation I” is received. Post adds to his “process” an instruction "(C) Stop”. He called such a process “type 1 ... if the process it determines terminates for each specific problem.” (Davis, 1965, p. 289ff)

- 1937: Alan Turing's paper *On Computable Numbers With an Application to the Entscheidungsproblem* reaches print in January 1937 (reprinted in Davis, 1965, p. 115). Turing’s proof departs from calculation by recursive functions and introduces the notion of computation by machine. Stephen Kleene (1952) refers to this as one of the "first examples of decision problems proved unsolvable".

- 1939: J. Barkley Rosser observes the essential equivalence of "effective method" defined by Gödel, Church, and Turing (Rosser in Davis, 1965, p. 273, "Informal Exposition of Proofs of Gödel's Theorem and Church's Theorem")
- 1943: In a paper, Stephen Kleene states that "In setting up a complete algorithmic theory, what we do is describe a procedure ... which procedure necessarily terminates and in such manner that from the outcome we can read a definite answer, 'Yes' or 'No,' to the question, 'Is the predicate value true?'."
- 1952: Kleene (1952) Chapter XIII ("Computable Functions") includes a discussion of the unsolvability of the halting problem for Turing machines and reformulates it in terms of machines that "eventually stop", i.e. halt: "... there is no algorithm for deciding whether any given machine, when started from any given situation, *eventually* ***stops***." (Kleene (1952) p. 382)
- 1952: "Martin Davis thinks it likely that he first used the term 'halting problem' in a series of lectures that he gave at the Control Systems Laboratory at the University of Illinois in 1952 (letter from Davis to Copeland, 12 December 2001)." (Footnote 61 in Copeland (2004) pp. 40ff)

2.11 Avoiding the halting problem

In many practical situations, programmers try to avoid infinite loops—they want every subroutine to finish (halt). In particular, in hard real-time computing, programmers attempt to write subroutines that are not only guaranteed to finish (halt), but are guaranteed to finish before the given deadline.

Sometimes these programmers use some general-purpose (Turing-complete) programming language, but attempt to write in a restricted style—such as MISRA C—that makes it easy to prove that the resulting subroutines finish before the given deadline.

Other times these programmers apply the rule of least power—they deliberately use a computer language that is not quite fully Turing-complete, often a language that guarantees that all subroutines are guaranteed to finish, such as Coq.

2.12 See also

- Busy beaver
- Generic-case complexity
- Geoffrey K. Pullum
- Gödel's incompleteness theorem
- Kolmogorov complexity
- P versus NP problem
- Termination analysis
- Worst-case execution time

2.13 Notes

[1] In none of his work did Turing use the word "halting" or "termination". Turing's biographer Hodges does not have the word "halting" or words "halting problem" in his index. The earliest known use of the words "halting problem" is in a proof by Davis (1958, p. 70–71):

> "Theorem 2.2 *There exists a Turing machine whose halting problem is recursively unsolvable.*
>
> "A related problem is the *printing problem* for a simple Turing machine Z with respect to a symbol S_i".

Davis adds no attribution for his proof, so one infers that it is original with him. But Davis has pointed out that a statement of the proof exists informally in Kleene (1952, p. 382). Copeland (2004, p 40) states that:

> "The halting problem was so named (and it appears, first stated) by Martin Davis [cf Copeland footnote 61]... (It is often said that Turing stated and proved the halting theorem in 'On Computable Numbers', but strictly this is not true)."

[2] Moore, Cristopher; Mertens, Stephan (2011), *The Nature of Computation*, Oxford University Press, pp. 236–237, ISBN 9780191620805.

[3] Stated without proof in: "*Course notes for Data Compression - Kolmogorov complexity*", 2005, P.B. Miltersen, p.7

[4] Börger, Egon. "Computability, Complexity, Logic". North-Holland, 1989. p. 121

2.14 References

- Alan Turing, *On computable numbers, with an application to the Entscheidungsproblem*, Proceedings of the London Mathematical Society, Series 2, Volume 42 (1937), pp 230–265, doi:10.1112/plms/s2-42.1.230. — Alan Turing, *On Computable Numbers, with an Application to the Entscheidungsproblem. A Correction*, Proceedings of the London Mathematical Society, Series 2, Volume 43 (1938), pp 544–546, doi:10.1112/plms/s2-43.6.544 . Free online version of both parts This is the epochal paper where Turing defines Turing machines, formulates the halting problem, and shows that it (as well as the Entscheidungsproblem) is unsolvable.

- Sipser, Michael (2006). "Section 4.2: The Halting Problem". *Introduction to the Theory of Computation* (Second ed.). PWS Publishing. pp. 173–182. ISBN 0-534-94728-X.
- c2:HaltingProblem
- B. Jack Copeland ed. (2004), *The Essential Turing: Seminal Writings in Computing, Logic, Philosophy, Artificial Intelligence, and Artificial Life plus The Secrets of Enigma,* Clarendon Press (Oxford University Press), Oxford UK, ISBN 0-19-825079-7.
- Davis, Martin (1965). *The Undecidable, Basic Papers on Undecidable Propositions, Unsolvable Problems And Computable Functions*. New York: Raven Press.. Turing's paper is #3 in this volume. Papers include those by Godel, Church, Rosser, Kleene, and Post.
- Davis, Martin (1958). *Computability and Unsolvability*. New York: McGraw-Hill..
- Alfred North Whitehead and Bertrand Russell, *Principia Mathematica* to *56, Cambridge at the University Press, 1962. Re: the problem of paradoxes, the authors discuss the problem of a set not be an object in any of its "determining functions", in particular "Introduction, Chap. 1 p. 24 "...difficulties which arise in formal logic", and Chap. 2.I. "The Vicious-Circle Principle" p. 37ff, and Chap. 2.VIII. "The Contradictions" p. 60ff.
- Martin Davis, "What is a computation", in *Mathematics Today*, Lynn Arthur Steen, Vintage Books (Random House), 1980. A wonderful little paper, perhaps the best ever written about Turing Machines for the non-specialist. Davis reduces the Turing Machine to a far-simpler model based on Post's model of a computation. Discusses Chaitin proof. Includes little biographies of Emil Post, Julia Robinson.
- Marvin Minsky, *Computation, Finite and Infinite Machines*, Prentice-Hall, Inc., N.J., 1967. See chapter 8, Section 8.2 "The Unsolvability of the Halting Problem." Excellent, i.e. readable, sometimes fun. A classic.
- Roger Penrose, *The Emperor's New Mind: Concerning computers, Minds and the Laws of Physics*, Oxford University Press, Oxford England, 1990 (with corrections). Cf: Chapter 2, "Algorithms and Turing Machines". An over-complicated presentation (see Davis's paper for a better model), but a thorough presentation of Turing machines and the halting problem, and Church's Lambda Calculus.
- John Hopcroft and Jeffrey Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, Reading Mass, 1979. See Chapter 7 "Turing Machines." A book centered around the machine-interpretation of "languages", NP-Completeness, etc.
- Andrew Hodges, *Alan Turing: The Enigma*, Simon and Schuster, New York. Cf Chapter "The Spirit of Truth" for a history leading to, and a discussion of, his proof.
- Constance Reid, *Hilbert*, Copernicus: Springer-Verlag, New York, 1996 (first published 1970). Fascinating history of German mathematics and physics from 1880s through 1930s. Hundreds of names familiar to mathematicians, physicists and engineers appear in its pages. Perhaps marred by no overt references and few footnotes: Reid states her sources were numerous interviews with those who personally knew Hilbert, and Hilbert's letters and papers.
- Edward Beltrami, *What is Random? Chance and order in mathematics and life*, Copernicus: Springer-Verlag, New York, 1999. Nice, gentle read for the mathematically inclined non-specialist, puts tougher stuff at the end. Has a Turing-machine model in it. Discusses the Chaitin contributions.
- Ernest Nagel and James R. Newman, *Godel's Proof*, New York University Press, 1958. Wonderful writing about a very difficult subject. For the mathematically inclined non-specialist. Discusses Gentzen's proof on pages 96–97 and footnotes. Appendices discuss the Peano Axioms briefly, gently introduce readers to formal logic.
- Taylor Booth, *Sequential Machines and Automata Theory*, Wiley, New York, 1967. Cf Chapter 9, Turing Machines. Difficult book, meant for electrical engineers and technical specialists. Discusses recursion, partial-recursion with reference to Turing Machines, halting problem. Has a Turing Machine model in it. References at end of Chapter 9 catch most of the older books (i.e. 1952 until 1967 including authors Martin Davis, F. C. Hennie, H. Hermes, S. C. Kleene, M. Minsky, T. Rado) and various technical papers. See note under Busy-Beaver Programs.
- Busy Beaver Programs are described in Scientific American, August 1984, also March 1985 p. 23. A reference in Booth attributes them to Rado, T.(1962), On non-computable functions, Bell Systems Tech. J. 41. Booth also defines Rado's Busy Beaver Problem in problems 3, 4, 5, 6 of Chapter 9, p. 396.
- David Bolter, *Turing's Man: Western Culture in the Computer Age*, The University of North Carolina Press, Chapel Hill, 1984. For the general reader.

May be dated. Has yet another (very simple) Turing Machine model in it.

- Stephen Kleene, *Introduction to Metamathematics*, North-Holland, 1952. Chapter XIII ("Computable Functions") includes a discussion of the unsolvability of the halting problem for Turing machines. In a departure from Turing's terminology of circle-free nonhalting machines, Kleene refers instead to machines that "stop", i.e. halt.
- Logical Limitations to Machine Ethics, with Consequences to Lethal Autonomous Weapons - paper discussed in: Does the Halting Problem Mean No Moral Robots?

2.15 External links

- Scooping the loop snooper - a poetic proof of undecidability of the halting problem
- animated movie - an animation explaining the proof of the undecidability of the halting problem
- A 2-Minute Proof of the 2nd-Most Important Theorem of the 2nd Millennium - a proof in only 13 lines

2.16 Text and image sources, contributors, and licenses

2.16.1 Text

2.16.2 Images

2.16.3 Content license