| Personal Information | Roll Number | 1914094 | 1914092 | 1914078 |
| --- | --- | --- | --- | --- |
| Name | Kaushal Binjola | Kevin Joshi | Devansh |
| Literature Information on | Paper Title | The origins of the halting problem | Problems with the Halting Problem | Approximations to the Halting Problem |
| Author of Paper | Salvador Lucas | Eric Hehner | Nancy Lynch |
| Year of Publication | 2021 | 2013 | 1973 |

**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.

For any program f that might determine if programs halt, a "pathological" program g, called with some input, can pass its own source and its input to f and then specifically do the opposite of what f predicts g will do. No f can exist that handles this case. A key part of the proof is a mathematical definition of a computer and program, which is known as a Turing machine; the halting problem is undecidable over Turing machines. It is one of the first cases of decision problems proven to be unsolvable. This proof is significant to practical computing efforts, defining a class of applications that no programming invention can possibly perform perfectly.

**The origins of the halting problem**

**Topic description:**

In this document, we will discuss the origins, formulation, and proof of the infamous Halting Problem. The *halting problem* is a prominent example of an undecidable problem and its formulation and undecidability proof is usually attributed to Turing's 1936 landmark paper. Copeland noticed in 2004, though, that it was so named and, apparently, first stated in a 1958 book by Martin Davis. From 1936 to 1958, when considering the literature of the field, no paper refers to any “halting problem” of Turing Machines until Davis' book. However, there were important preliminary contributions by Church, for whom termination was part of his notion of computation (for the *λ*-calculus), and Kleene, who essentially formulated, in his 1952 book, what we know as the halting problem now.

**Concept behind the paper:**

To define the concept of the problem simply we can take an example of programming languages.

In teaching courses about programming languages, program analysis, and software engineering, the halting problem is frequently invoked to justify the *well-known piece of folk-lore among programmers that it is impossible to write a program which can examine any other program and tell, in every case, if it will terminate or get into a closed loop when it is run.*

Besides its obvious interest for specialists in program termination, and its use to provide a first understanding of undecidability issues, other authors acknowledge the halting problem as “one of the most philosophically important theorems of the theory of computation”.

Although *“it is often said that Turing stated and proved the halting problem in his 1936 paper”,* it was named and originally stated by Martin Davis.

Recent books about Turing's works and contributions already acknowledge this fact. Some papers about program termination now include some mild warnings about the usual association of the halting problem to Turing. Still, many papers and textbooks refer to the definition and proof of the undecidability of the halting problem to ‘**On computable numbers, with an application to the Entscheidungsproblem**’ by Turing.

We agree with Copeland that crediting the authorship of the definition and proof of the unsolvability of the halting problem to Turing is not correct. This can be seen by 3 different arguments:-

1. A motivational point of view, where we claim that, from the material into ‘On computable numbers, with an application to the Entscheidungsproblem’ by Turing, we would rather say that Turing was not especially interested in halting machines; we believe that this is the reason why, as Copeland observes, he pays no attention to this notion in his 1936 paper; then
2. We show that the decidability problems considered by Turing are conceptually different from the halting problem, and, finally,
3. From a bibliographical point of view, we notice the absence of references to any “halting problem” in the considered bibliography from ‘On Computable..’ to the first publication referring to a “halting problem”.
4. In his 1952 book ‘**Introduction to Metamathematics**’ Kleene provides a statement which we easily recognize as what is called a “halting problem” today.

We will see the above points in detail below and show how Martins is to be credited for the present definition of the halting problem.

**Problem/Solution discussed:**

**1. Turing machines and Turing’s notion of computability-**

A Turing is a device that operates on an infinite *tape* arranged as an infinite sequence of possibly empty *cells*. A *head* is able to *scan* the tape by reading and writing (or *printing*) symbols; it moves step by step along with the tape. The machine is able to adopt finitely many *statesq*1, ..., *qn* (*m-configurations* in Turing’s terminology). The *operations* performed on the tape (i.e., printing symbols and/or moving to the left or to the right), and also possible changes of state, are determined by the so-called configuration *qi*, *sj* of the machine when operating in the state *qi* and the scanned symbol in the *j*-th cell of the tape is *sj*. The a-machines, usually called (simple) Turing Machines

Turing’s notion of computation pays no attention to any halting behavior. In the subsection about *computing machines*, i.e. given an automatic machine, he writes on pg 232:

*if the machine is supplied with a blank tape and set in motion, starting from the correct initial m-configuration, the subsequence of the symbols printed by it which are of the first kind will be called the sequence computed by the machine.*

The notions of *circular* and *circle-free* machines are given in pg. 233:

*If a computing machine never writes down more than a finite number of symbols of the first kind, it will be called circular. Otherwise, it is called circle-free.*

At first sight, the previous sentence may suggest that circular machines always halt. However, the clarification that immediately follows denies this:

*A machine will be circular if it reaches a configuration from which there is no possible move, or it goes on moving, and possibly printing symbols of the second kind.*

The fact that Turing's partition of his machines pays no special attention to any halting behavior suggests that at least in ‘On computable ..’, *he was not particularly interested in investigating halting machines*. Furthermore, he makes explicit that only (subsequences of) infinite sequences of figures (computed by circle-free machines) are considered as *computable* ‘On computable ..’, page 233:

*A sequence is said to be computable if it can be computed by a circle-free machine.*

Circular machines may halt, but, by definition, circle-free machines *never stop*.

**2.** **Martin Davis' description of Turing machines and computations**

Since the notion of the computed sequence includes taking any subsequence of figures printed by a machine, computing finite sequences is possible but the exact definition of such a sequence is not intrinsically provided by the machine operation. An “external observer” is required to extract the subsequence. This is in sharp contrast with the standard, algorithmic, idea of a computation by a TM, which requires that the machine halts, thus providing a completely defined finite sequence as the one which is obtained when the machine halts, cf. M.D. Davis Computability and Unsolvability, Definition 1.9:

*By a computation of a Turing machine, M is a finite sequence of instantaneous descriptions α1,…,αp such that αi→αi+1 for 1≤i<p and such that αp is terminal with respect to M.*

The key point here is that the last instantaneous description αp must be *terminal*, i.e., no transition is possible from it.

In other words,

“*The machine interprets the absence of an instruction as a stop order”* When reaching αp .

This is the usual notion of computation with TMs today.

**3.** **Decision problems about Turing machines**

In his paper, Turing enunciates two problems about his machines (now

satisfactoriness and printing problems) and proves them undecidable.

### **The satisfactoriness problem**

The first problem considered by Turing is given on page 247, as follows:

*Is there a machine* D *which, when supplied with the S.D. of any computing machine*

M *will test this S.D. and if* M *is circular will mark the S.D. with the symbol ‘u’ and if it*

*is circle-free will mark it with ‘s’?*

Here,‘u’ and ‘s’ represent an *unsatisfactory* or *satisfactory* verdict of D about M being

circle-free.

### **The printing problem**

### The second problem, considered on page 248, is what Turing used to show the

### [*Entscheidungsproblem*](https://www.sciencedirect.com/topics/computer-science/entscheidungsproblem) impossible:

*there can be no machine* E *which, when supplied with the S.D. of an arbitrary*

*machine* M *will determine whether* M *ever prints a given symbol (0 say)*.

Davis uses *printing problem* (PRINT in the following) to refer to it. In the following, for

each machine M, we say that Print(M) is *true* iff M prints a given symbol, (e.g., 0)

during its (finite or infinite) execution (and often say that M is a *printing* machine).

For instance, both Print(PC1) and Print(PC2) are *false*, but Print(PC3) and

Print(PCF) is *true*.

**Davis' definition of the halting problem.**

**a.** **Halting Problem**

According to Davis' formulation,

The halting problem for a TM M aims *to determine whether or not* M*, if placed in a given initial state, will eventually halt.*

This is the standard understanding of the halting problem for TMs (HALT in the following). Davis proves it undecidable. In the following, we say that Halt(M) is *true* if M halts when placed in a given initial state (and often says that M is a *halting* machine). For instance, Halt(PC1) is *true*, but Halt(PC2), Halt(PC3), and Halt(PCF) is *false*.

### **Relationship between *printing* and *halting* problems**

In his book, Davis first introduces the *halting* problem and then the *printing* problem. He does not mention whether Turing introduced one or the other. When introducing the printing problem, he calls it “*a related problem*” (to the previously introduced halting problem). The presentation he gave of the printing problem in his book i.e.,

*To determine, of a given instantaneous description of* M*, whether or not there exists a sequence* α1,…,αk *of instantaneous descriptions such that* α=α1*,* αj−1→αj *for* 1<i≤k*, and* αk *contains the symbol s.*

Clearly fits Turing's definition as given above (provided that *s* is 0, although Turing's formulation, “*prints a given symbol (0 say)*

**4. Bibliographical analysis**

The (necessarily incomplete) analysis of the literature about computability and undecidability for the period from 1936 to 1958 shows that, apparently, the term “halting problem” was not used before the publication of M.D. Davis Computability and Unsolvability. A number of journals and repositories in the fields of mathematics, logic, and computer science with facilities to search for (prefixes of) words in their archives were considered and ‘halt’ tried on the corresponding search devices for the period 1936-1958.

**Conclusion:**

The subject of Turing's paper was “*ostensibly the computable numbers*”. He defines a number as computable if “*its decimal part can be written down by a machine*”; and points to numbers like *π* (with infinitely many decimals without any repetition pattern), as computable. Thus, if Turing's focus was on infinitary computations, it is not surprising that investigating halting machines was not a priority.

We can henceforth, successfully conclude that it was M.D. Davis who came up with the present used the definition and idea behind the halting problem, and he should be credit for it.

**Summary:**

When introducing his a-machines, Turing was not interested in halting machines and his notion of computation focused instead on the generation of infinite sequences of figures. His undecidability results were according to this.

Davis' notion of computation required a halting behavior of the machines. It is not surprising, then, that Davis formulated the halting problem and proved it undecidable. Strikingly, the last sentence of M.D. Davis Computability and Unsolvability :

*It might also be mentioned that the unsolvability of essentially these problems was first obtained by Turing probably contributed to give credit on the halting problem to Turing.* However, as discussed in the previous sections, and summarized through Facts 1–4, the conceptual differences between Turing's decision problems and the halting problem suggest that this claim just honors Turing's pioneering work in posing and solving relevant problems in Computer Science, including the undecidability of the satisfactoriness and printing problems. However, the halting problem is not among the contributions of Turing in.

**References:**

* Salvador Lucas, The origins of the halting problem, Journal of Logical and Algebraic Methods in Programming, Volume 121, 2021, 100687, ISSN 2352-2208, https://doi.org/10.1016/j.jlamp.2021.100687.

**Problems with the Halting Problem**

**Topic Description:**

Either we leave the definition of the halting function incomplete, not saying its result when applied to its own program, or we suffer inconsistency. If we choose incompleteness, we cannot require a halting program to apply to programs that invoke the halting program, and we cannot conclude that it is incomputable. If we choose inconsistency, then it makes no sense to propose a halting program. Either way, the incomputability argument is lost. So this topic focuses on trying to understand the problems faced during the resolution of the halting problem that leads to the conclusion of inconsistency being the solution and trying to find out a mathematical derivation to either solve the halting problem or derive why the halting problem is unsolvable.

**Concept behind the paper:**

The concept behind the paper was to understand in depth the reason due to which the halting problem arises and trying to formulate a solution using the knowledge gained over the years since the halting problem was first recognized if not trying to derive an equation that proves that the halting problem is not computable and hence it cannot be solved.

**Problem/Solution discussed:**

In 1936 Alan Turing wrote a paper introducing a computing machine that we now call a “Turing Machine”. He showed how it can be programmed, and how different programs give the machine different behaviors, or in his terminology, create different machines. He showed how one such machine, a “Universal Turing Machine”, can be given a description of any Turing Machine (a program), and then simulate the operation of that Turing Machine (execute the program). That work remains an important part of the foundation of computer science. Page 247 of that paper is proof that a certain problem that we now call the “Halting Problem” cannot be solved by computation.

So the whole paper discusses and tries to revisit the proof of the problem to understand why the halting problem was unsolvable back then and whether it is solvable now.

Notion and terminology used:-

For the purpose of the paper, we need a programming language that includes the following three kinds of programming statements:

x:= E

if B then P else Q

while B do P

These are standard assignment, conditional, and loop statements found in most modern

programming languages (except perhaps for minor syntactic differences). In the assignment statement, x is a program variable, and E is an expression of the same type as x . In the next two, B is a boolean expression, and P and Q are programming statements. The two boolean values are T (true) and ⊥ (false). By “boolean expression” I mean an expression whose overall type is boolean; subexpressions within it are not restricted to type boolean. Any programming language will include only a restricted choice of types and operators; for our purpose, all we need is a call, and some way to return a result. I refer to any program fragment (statement or expression) as a “program”.

The proof discussed in the paper:

*Define function h: P → B , where P is the data type of texts that represent programs,*

*and B is the boolean data type, so that when h is applied to a text representing a*

*program whose execution terminates, the result is T , and when applied to a text*

*representing a program whose execution does not terminate, the result is ⊥ . Here are*

*two examples:*

*h (“ x:= 0 ”) = T*

*h (“ while T do x:= 1 ”) = ⊥*

*Assume that h is computable, and that a program H for it has been written. Then*

*H (“ x:= 0 ”) = T*

*H (“ while T do x:= 1 ”) = ⊥*

*Define program text D (for diagonal) as follows:*

*D = “ if H (D) then while T do x:= 1 else x:= 0 ”*

*We place the definitions of H and D in a dictionary (or library) of definitions. When*

*identifiers (like H and D ) occur within a program, they are interpreted (or understood,*

*or given meaning) by looking them up in the dictionary.*

*Argue: If H (D) is T , then D represents a program that is equivalent to*

*while T do x:= 1 , whose execution does not terminate, and so H (D) is ⊥ . And if*

*H (D) is ⊥ , then D represents a program that is equivalent to x:= 0 , whose execution*

*does terminate, and so H (D) is T . We have an inconsistency.*

*Conclude: program H does not exist; function h is not computable.*

Many other such derivation are discussed mathematically in the paper such as “Calumation Problem”, “Rusell’s Barber”, “Oracle halting problem”, “Specification halting problem”, “Mimic halting problem”, “Simple halting problem”...

All of these derivations had a common characteristic of having a function such as the H function in the above derivation that could not be resolved and that function being the reason for the halting problem. The function H in some cases was a recursive function in itself or called another recursive function or contained a loop that had potential or ended up being an infinite loop. In such cases various derivations were made all over the research paper that ended in a conclusion that to know whether the function H is infinite or not there is no way to initially conclude that even before execution of the program. The computation of whether a program is going to end up being infinite or not is not computable with 100% accuracy without execution of the program and if the program is executed it will end up in an infinite loop.

**Conclusion:**

The halting problem proves the fact that the programs that will endup in an infinite loop are going to be inconsistent and cannot be pro computed to avoid them, an inconsistent program can only be identified once it is executed and based on the parameters for eg. parameters for a recursive function if unchanged will result in an infinite loop but the fact that the parameters are going to be unchanged cannot be computed without execution of the actual program. There are many other cases that have arised in today's time when compared to turing’s time such as almost all programs now a days are interactive programs that depend on some sort of input either from a program or a user and such inputs are unidentifiable and hence whether the program will be inconsistent for any of inputs from the pool of infinite inputs cannot be computed without execution. And hence the halting problem cannot be solved or avoided by the system.

**Summary:**

Halting problem still remains a problem that cannot be solved after studying the problems that are faced in solving the halting problem. The conclusion derived from it is that halting problem is majorly caused due to the inconsistency in the program that arises majorly due to the dependency of the program on the input which cannot be computed without execution of the program and once the program is executed and it is inconsistent it will end up in an infinite loop.

**References:**

* E.C.R.Hehner: Problems with the Halting Problem, COMPUTING2011 Symposium on 75 years of Turing Machine and Lambda-Calculus, Karlsruhe Germany, invited, 2011 October 20-21 Advances in Computer Science and Engineering v.10 n.1 p.31-60, 2013

**Approximations to the Halting Problem**

**Topic Description:**

* In 1931, the logician KURT GODEL constructed a mathematical predicate that could neither be proven nor falsified. In 1936, ALAN TURING introduced and showed the Halting Problem H to be undecidable by a Turing machine. This was considered a strengthening of Godel’s result regarding that, at this time and preceding AIKEN’s Mark I and ZUSE’s Z3, the Turing machine was meant as an idealization of an average mathematician.
* Nowadays the Halting Problem is usually seen from a quite different perspective. Indeed with the advent of an increasing reliance on high-speed digital computers and huge pieces of software running on them, source code verification or at least the detection of stalling behavior becomes ever more important.
* In fact, by RICE’s Theorem, this is equivalent to many other real-world problems arising from goals like automatized software engineering, optimizing compilers, formal proof systems, and so on. Thus, the Halting problem is a very practical one that has to be dealt with in some way or another. One direction of research considered and investigated the capabilities of extended Turing machines equipped with some kind of external device solving the Halting problem.
* While the physical realizability of such or other kinds of super-Turing computers is questionable and in fact denied by the Church-Turing Hypothesis, the current field of Hypercomputation puts this hypothesis into question. On the theoretical side, these considerations led to the notion of relativized computability and the Arithmetical Hierarchy which have become standard topics in Recursion Theory.
* Since this problem cannot be solved, we find approximate solutions for the halting problem.

**Concept behind the paper:**

* In decision problems, a notion of the approximate solution has been established in Property Testing. Here for input x ∈ Σ n, the answer “x ∈ L” is considered acceptable even for x not belonging to L provided that y ∈ L holds for some y ∈ Σ n with (edit or Hamming) distance d(x,y) ≤ εn. Observe that this notion of approximation strictly speaking refers to the arguments x to the problem rather than the problem L itself. Also, any program source x is within constant distance from the terminating one y obtained by changing the first command(s) in x by a halt instruction.
* Average case analysis is an approach based on the observation that the hard instances which make a certain problem difficult might occur only rarely in practice whereas most ‘typical’ instances might turn out as easy.
* So, although for example NP-complete, an algorithm would be able to correctly and efficiently solve this problem in, say, 99.9% of all cases while possibly failing on some few and unimportant others. In this example, ε = 1/1000 is called the error rate of the problem under consideration with respect to a certain probability distribution or encoding of its instances.
* Such weakenings have previously been mainly applied in order to deal with important problems where the practitioner cannot be silenced by simply remarking that they are NP-complete, that is, within complexity theory. However the same makes sense, too, for important undecidable problems such as Halting: even when possibly erring on, say, every 10th instance, detecting the other 90% of stalling programs would have prevented many buggy versions of a certain operating system from being released prematurely.

There have been several propositions and proofs for the same trying to sharpen the approximation where using the results from the proofs, some propositions are termed false and others can not find the exact solution and reach an approximation for the same.

**Problem/Solution discussed:**

Halting means that the program on certain input will accept it and halt or reject it and halt and it would never go into an infinite loop. Basically halting means terminating. So can we have an algorithm that will tell us whether the given program will halt or not. In terms of a Turing machine, will it terminate when run on some machine with some particular given input string.

We cannot design a generalized algorithm that can appropriately say that a given program will ever halt or not. The only way is to run the program and check whether it halts or not.

One way to define the Halting problem with respect to programming is:

* A G¨odelization ϕ is a sequence of all partial recursive functions s.t.

– there exists a partial universal program u with ϕu(hi,xi) = ϕi(x) (UTM)

– and a total program s with ϕs(hi,xi) (y) = ϕi(hx,yi) (SMN)

– for a bijective computable function h<·,·>i : Σ ∗ ×Σ ∗ → Σ ∗ or h<·,·>i : N×N → N. called pairing function. The Halting problem for ϕ is Hϕ = {hi,xi: x ∈dom(ϕi)}

* The Halting problem is sometimes alternatively defined as the task H˜ ϕ of deciding whether a given program i terminates on the empty input, that is, whether λ ∈ dom(ϕi); or the question whether i ∈ dom(ϕi).
* The halting ratio states that neither nearly all programs halt nor do nearly all of them stall.

**Conclusion:**

* Since the Halting problem is of practical importance yet cannot be solved in the strict sense, we considered the possibility of approximating it. Similar to the average-case theory of complexity, this depends crucially on the encoding of the problem, that is here, the programming system under consideration.
* Many practical programming languages lacking density in fact do admit such an approximation with asymptotically vanishing relative error for the simple reason that the fraction of syntactically incorrect instances tends to 1. This was exemplified by combinatorial analysis of the Turing-complete formal language BF.
* Here and in similar cases, the question for approximation of the Halting problem is equivalent to a mere syntax check and thus becomes trivial and vain. On the other hand, considering only syntactically correct sources was established to yield an efficient and dense programming system in the case of BF. For any such system, we proved a universal constant lower bound on relative approximations to the Halting problem even in the weak io-sense.

Hence from this paper, we can conclude that there is no algorithm to solve the halting problem but using the proposed definitions and their respective proofs, we can approximate it to an extent.

**Summary**

We cannot design a generalized algorithm which can appropriately say that a given program will ever halt or not. The only way is to run the program and check whether it halts or not. Since there is no algorithm to solve the halting problem but using the proposed definitions and their respective proofs, we can approximate it to an extent.

**References:**

* Nancy Lynch, Approximations to the Halting Problem, Mathematics Department, University of Southern California, Los Angeles, California 90007 Received January 27, 1973

**SUMMARY BASED ON ABOVE THREE PAPERS**

* Turing Machine and Halting problems are part and parcel of the same thing, so we see how the credits of halting problem are naturally credited to Alan Turing, it is a fact that the present followed the definition of the halting problem was first given and defined by Martin Davis. Turing was not interested in halting machines and his notion of computation focused instead on the generation of infinite sequences of figures. His undecidability results were according to this.

Davis' notion of computation required a halting behavior of the machines. It is not surprising, then, thatDavis formulated the halting problem and proved it undecidable**.**

* The problems faced in halting problem when studied mathematically resulted in the conclusion that the halting problem cannot be resolved since the inconsistency in the problem cannot be pre-identified or computed before execution of the program and when the program is executed, it ends up in an infinite loop.
* We cannot design a generalized algorithm that can appropriately say that a given program will ever halt or not. The only way is to run the program and check whether it halts or not. Since there is no algorithm to solve the halting problem but using the proposed definitions and their respective proofs, we can approximate it to an extent.