Macalester Journal of Philosophy

Volume 5
Issue 1 *Spring 1994*

Article 4

2-17-2011

The Halting Problem

Tracy D. Banitt
Macalester College

Tracy D. Banitt

"The Halting Problem"

Since Turing's Thesis was defined in the mid-1930s, many interesting questions have been asked about Turing machines and their applications. One of these questions is the halting problem. In this paper I will give a brief explanation of Turing Machines and Turing's Thesis. I will also define and prove that the halting problem is undecidable. Lastly, this paper will show some applications of the results from the halting problem.

Turing's Thesis

If the reader is to understand the halting problem and its applications, an explanation of Turing Machines and the history of Turing's Thesis is helpful. This section of the paper will start off by giving some background information on Turing and explaining what the thesis states. Next, this section will briefly define what a Turing Machine is and how it can be used.

Alan Turing was born in Great Britain in June, 1912. He was raised with a traditional British upper-middle-class education and later attended King's College at Cambridge University. He graduated with distinction in mathematics in 1934 and was elected to a Fellowship of King's College in the following year.

In 1935 Turing attended a course in mathematics which introduced him to the Entscheidungsproblem, which was unsolved at the time. Turing was attracted to the problem and in April, 1936, submitted a paper entitled "On Computable Numbers, with an Application to the Entscheidungsproblem." In this paper Turing introduced the idea of the Turing machine; his thesis was soon to follow.

Turing's Thesis is extremely simple, and states that any computation that can be carried out by mechanical means can be performed by some Turing Machine. It is important to remember that Turing's Thesis is only a thesis. No one has proven Turing's Thesis to be correct because to do so we would have to define what is meant by "mechanical means." Yet if we regard the thesis as a definition and assume that the definition is sufficient enough to cover everything computers can do now and will ever be able to do, then the thesis is acceptable.

Now that Turing's Thesis has been explained, a brief definition of what a Turing Machine looks like can be given. Although there are infinitely many Turing Machines, this paper will describe a general Turing Machine. A Turing Machine can be thought of as a black box which has a read-write head that points at a tape. The tape is infinite in both directions and is divided into cells, each of which is capable of holding one symbol. The tape holds the input for the Turing Machine, and the Turing Machine is allowed to read information from the tape and write information to the tape.

The black box is the actual Turing Machine and is expressed as a tuple, TM = {Q, E, G, s, q0, b, F}. Q is the finite set of internal states that the Turing Machine can be in. The symbol E represents the finite set of symbols the Turing Machine recognizes as input, called the input alphabet. The set of all symbols which are allowed to be on the tape is represented by the symbol G. S is the transition function which causes the Turing Machine to 'move'. The state the Turing Machine starts in is q0. The symbol

b represents a special tape symbol called "the blank." Lastly, F is the finite set of all final states; in other words, F is the set of all states that cause the Turing Machine to halt.

A "move" on a Turing Machine is defined using the S transition function and is written thus: S(qi, G) = (qj, G, {L or R}). This transition statement says that a Turing Machine M, in state qi looking at a particular tape symbol in G, will change its state to qj and replace the symbol on the tape with the new symbol from G. The {L or R} represents the direction the read-write head is to move when the operation is complete (L means left and R means right). Some Turing Machines will have a stay option, s, which allows the Turing Machine to not move the read-write head on a specific transition. This feature is not necessary since it adds no power to the Turing Machine.

A Turing Machine contains many transition functions. Everything a Turing Machine can do is defined by these transition functions. If a Turing Machine encounters a situation on the tape where it is looking at an unknown symbol to its specific state, it is said to "blow up." This is my own little term, since a Turing Machine could not actually "blow up." It is important to note that a Turing Machine does not halt when it blows up. The Turing Machine simply is stuck and doesn't know what to do. It has not halted because it is not in a final state. A Turing Machine halts when it does not know what to do next and has reached a final state.

Now that Turing's Thesis and Turing Machines have been briefly defined, we are ready to attack the halting problem.

Halting Problem

This section of the paper will define the halting problem and prove that it is undecidable. There are two different proofs in this section, each using proof by contradiction. The first is straightforward and the second uses the notion of recursively enumerable languages. First, a definition of the halting problem is needed.

The halting problem asks, given the description of a Turing Machine M and an input w, does M starting in the start state q0 perform a computation that eventually halts? In other words, will M eventually stop in a final state? The answer to this question is no. The following proof is straightforward and clear, I hope.

We start by assuming that the halting problem is decidable. This is our assumption that will be negated at the end when a contradiction arises. If the halting problem is decidable, we can then create a Turing Machine that will be able to solve the halting problem. Call this Turing Machine H. We will further define H to be the Turing Machine that takes as input the description of M and the input string w and halts in state qy if M halts on w, or halts in state qn if M does not halt on input w.

From H we build a new Turing Machine H' which acts exactly like H except when H goes into state qy, H' is going to go into an infinite loop. This is very easy to do; we simply add three transition functions and two new states to H. The first sends H to state qa if H is in state qy. The second sends H to qb if H is in qa. The last sends H to qa if H is in qb. These three transition statements do not care what symbol the read-write head is looking at. (See Figures A and B.)

Figure A

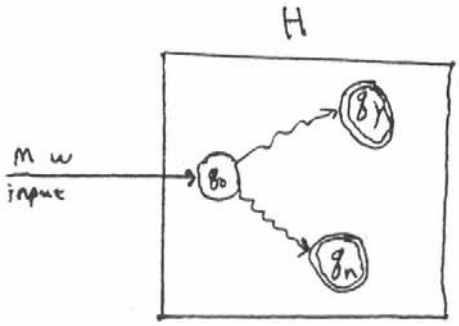
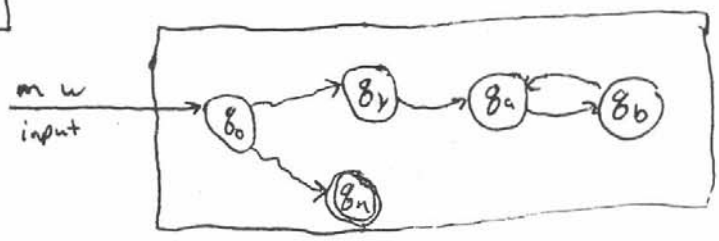


Figure B



It is easy to see that H' is not that different from H. With input M and w, the Turing Machine H' will go into an infinite loop if M halts on w; or if M does not halt on w, H' will halt in state qn. Now, from H' we create another new Turing Machine H" which takes as input the description of a Turing Machine, copies that description, and then behaves exactly like H'. The reason that H" copies the description of the Turing Machine is because we need to get the input into the form; description of M and an input string w. H" copies the description so that the input is now of the form: description of M description of M, where the second description of M is treated as if it were w. In summary, H" takes as input a description of Turing Machine M; it goes into an infinite loop if M halts with a description of itself as input, and goes to state qn if M does not halt with itself as input.

The final step in this proof is the contradiction. Since H" is also a Turing Machine, we can take a description of it and send it through H". This means that H" will halt in state qn if H" does not halt, and H" will not halt if H" halts. This is obviously nonsense, and we can now negate our assumption, which means there does not exist a Turing Machine which can solve the halting problem. Therefore, the halting problem is undecidable.

The second proof uses recursively enumerable languages. The proof starts out by saying that if the halting problem is decidable, then every recursively enumerable language would be recursive. This is not true, and thus the halting problem is not decidable. The proof is as follows.

First, we let L be a recursively enumerable language. We also say that M is a Turing Machine that accepts the language L. That is, whenever a string of characters is in the language L, M will halt in some state qy, and if the string of characters does not belong to the language L, M will not halt at all.

Next, we again assume that H is a Turing Machine that solves the halting problem. We apply the description of M and the string w to H; if H says M does not halt on string w, then w is not in L. If H says yes, then w is in L, so then apply w to M. But we know that M must halt, so M will eventually tell us whether w is or is not in L. This means we have a membership algorithm, if we have a membership algorithm L recursive. Yet there are recursively enumerable languages that are not recursive. Therefore, we have a contradiction and can negate our assumption that there exists a Turing Machine that can solve the halting problem. Once again the halting problem is proven undecidable.

These two proofs clearly show that the halting problem is undecidable. The next section of this paper will show how we can use these results to answer questions about other problems' decidability.

Applications

Using the results from the halting problem, we can prove many other undecidable problems by reducing them to the halting problem. I will call this method of proof the reduction method. Reducing one problem to the halting problem is another way of saying we are going to use the logical rule MTT. We start by saying that if problem A is decidable, then the halting problem is decidable. Since we know the halting problem is undecidable, we can negate our assumption that problem A is decidable. This means that A is undecidable.

A good example of the reduction proof method is the blank tape halting problem. This problem asks, given a Turing Machine M and a blank tape, can we determine if M will ever halt using the blank tape as input? It looks as if we could prove this problem to be undecidable in the same way we proved the original halting problem (and actually we could), but it is a good example to introduce the reduction method. The proof that the blank tape halting problem is undecidable follows.

First we assume we have a Turing Machine M' which can solve the blank tape halting problem. Next, take any arbitrary Turing Machine M and any input string w. Construct from M a new Turing Machine M'. M' will start with the blank tape, write the string w on the tape, and then position the read-write head at the beginning of string w. M' will then change its state to the start state q0 and act just like M would on w. Clearly, M' will halt on a blank tape if and only if M will halt on the input string w. But if we can decide whether M will halt on input string w, we have answered the halting problem. We already know that the halting problem is undecidable, so we have a contradiction and can negate our assumption. The conclusion is that the blank tape halting problem is undecidable.

Another example of the reduction method is the state entry problem. This problem asks if, given a Turing Machine M and an input string w, will M ever enter the specific state qi during the computation process? This problem can also be proven undecidable by reducing it to the halting problem. The proof is as follows.

First, we assume the state entry problem is decidable. We modify the original machine M to get a Turing Machine M', which acts exactly as M does except if M halts on the string w then M' will halt in the state qi, and if M does not halt on w, then M' will halt in the state qn. Again we see that if we had a way of deciding the state entry problem, we would also have a way of deciding the halting problem. Since

we know that the halting problem is undecidable, we can conclude that the state entry problem is also undecidable.

These two examples help show the usefulness of the halting problem. Proving that these two problems are undecidable would be much more difficult if we could not simply reduce them to the halting problem.

In a broader picture, proving the halting problem is undecidable answers many larger questions. For example, if the halting problem was proven decidable, then someone would no doubt try to create a Turing Machine that could solve the halting problem. With a Turing Machine that can solve the halting problem, it is not hard to imagine a computer program that can decide whether any given computer program will stop on a specific input. The next step after this is writing a program that can decide if a specific computer program is correct for a certain task. After this, the next logical step is to create a computer program which can generate correct computer programs for a specific problem. This would not go over well, since many computer programmers would be out of jobs. Yet, since the halting problem is undecidable, this does not seem a possibility. If artificial intelligence is considered, then this last statement seems false. This issue is definitely worthy of further study, but since this paper is about the halting problem and Turing Machines, I will not go into this topic.

In conclusion, proving the halting problem is undecidable has many different applications in the world. The two problems discussed in this paper are part of a large set of problems that the halting problem has helped to solve. I doubt if Alan Turing knew what the consequences of his paper would be when he wrote it in 1936, but I'm sure he is pleased with the results. Turing's Thesis has given the computer science world a great tool for solving problems.

Bibliography

Cohen, Daniel I.A., *Introduction to Computer Theory.* New York: Wiley, 1986.

Davis, Martin, *Computability, Complexity, and Languages.* New York: Academic Press, 1983.

Herken, Rolf, *The Universal Turing Machine.* New York: Oxford Univ. Press, 1988.

Hodges, Andrew, *Alan Turing: The Enigma.* New York: Simon & Schuster, 1983.

Linz, Peter, *An Introduction to Formal Languages and Automata.* Lexington, MA: D.C. Heath, 1990.