Turing Machines and the Halting Problem

Alvin Li

Introduction

In 1936 Alan Turing conceived the Turing Machine, a model of the computer that is able to simulate any algorithm that could theoretically be performed by a computer. Since it perfectly simulates a computer, it could be used to answer any question about the theoretical limits and possibilities of a classical computing device. One of the more famous results that came out of the usage of Turing Machines was the undecideability of the Halting Problem, which states that no Turing Machine could determine if another Turing Machine halts on a certain input. This result, in turn, played a large role in solving many problems regarding decideability and computability.

Turing Machines

Turing machines are the simplest theoretical models of a computer. Turing machines can be represented physically as a head attached to an operating box which examines an infinite tape of discrete cells. The head can move the tape to the right or left, read the symbol in the current cell, or change the symbol in that cell.

A function is defined by a Turing Machine M as follows:
Let M's tape alphabet be Σ, which is the list of symbols appearing in its tape. Let x be a string formed from Σ. M parses the tape without interference, and if M halts and a string y is found on the tape, then y is M's output corresponding with the input x. Note that M does not necessarily halt for all x. The function of M can be written as

$$f : \Sigma^* \to \Sigma^*$$

Where Σ^* is the set of strings over Σ. This set acts as the co-domain and range of the function.

A Turing Machine can be represented by its *program*, which is a set of quintuples (q, s, q', s', d) where q is the state of M, s is the symbol currently being inspected by the head, q' is the state that M is about to enter, s' is the symbol in place of s after it has been parsed, and d is the direction in which M moves after inspecting s.

At any point in time, M can be in a state q which comes from a finite set Q that includes an initial state q_0 and a halting state q'. Each state carries a set of instructions which govern how the Turing Machine behaves in response to what is read on the input tape. These instructions include moving to the left or right, writing to the tape, or halting. If the state is not a halting state, it must also carry instructions to transition to another state. These state transitions can be represented with a state-transition diagram.

With this input and output system, a Turing Machine is able to compute anything that is computable by a modern computer. In fact, it can be proven that a Turing Machine's computational power is unaltered by the number of states available or the number of symbols in its tape alphabet. This model is usually the one chosen to represent a theoretical computer or computer program in computation theory because of its robustness.

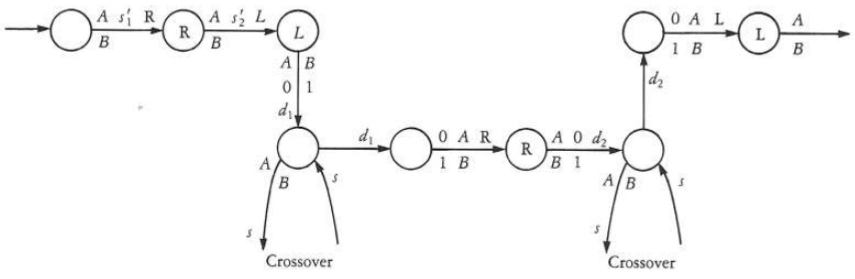

Figure 1: An example of a state-transition diagram.

The Halting Problem

The Halting Problem asks if there exists a Turing Machine that can, given a description of the program of another Turing Machine and its input tape, whether that Turing Machine halts on its input. In 1936, Alan Turing proved that no such Turing Machine exists.

Proof. Suppose for the sake of contradiction that a Turing Machine T_H which can determine halting exists. The input tape consists of a description of T and its input tape t. If T halts on t, then T_H outputs "yes" in some form and enters a halting state q_h. Otherwise, if T does not halt on t, then T_H outputs "no" and also enters some other halting state q_k.

Construct a Turing Machine T'_H as a modification of T_H, where the original "yes" halting state q_h is replaced with another state q_n which transitions only to itself for all possible inputs. The "no" state q_k stays as it is. Since q_n isn't a halting state and only maps to itself, there is no way of transitioning into a halting state from q_n. Thus, T_H does not halt when the input Turing Machine halts, and it essentially enters into an infinite loop.

Consider what happens when a description of T'_H, dT'_H, is inputted into T'_H itself. Then, it asks whether or not T'_H halts on itself as input. Suppose it did, then T'_H outputs "yes" and enters into q_n, where it doesn't halt. This implies that T'_H does not halt on itself, which is a contradiction. Now, suppose T'_H does not halt on itself. Then T'_H would output "no" and enter into q_h, which is a halting state. This implies that T'_H halts on itself, supplying another contradiction. Since all possible outputs of T'_H result in contradictions, then it must be the case that T'_H doesn't exist in the first place. Therefore, T_H also cannot exist, as desired.

The logic of this is quite simple, and could be implemented in Python:

```python
def halts(func,arg):
    pass

def halts2(func):
    if halts(func,func):
        while True:
            pass
    else:
        return

halts2_descr = """
def halts2(func):
    if halts(func,func):
        while true:
            pass
    else:
            return
"""
halts2(halts2_descr)
```

In the code, the function `halts()` represents the program of T_H, `halts2()` represents the program of T'_H, and `halts2_descr` is its description, dT'_H. We do not write a body for `halts()` because it was proven that no T_H actually exists. The description of T'_H can be represented with a string because it can be passed into another function and holds the same information as `halts2()`, which serves the same purpose as dT'_H. When passed into another function, it can be run with `exec()`. Aside from these technical details, the logic follows from the proof that was supplied.

Applications

The undecidability of the Halting Problem has many consequences on other decidability problems. Many more general decidability problems can be proven by reducing it to a subproblem of the Halting Problem. Then, since we know the Halting Problem to be undecidable, conclusions can be drawn. We say that a decidability problem (bascially a yes or no question) is undecidable if there doesn't exist a Turing machine which halts on all possible inputs when computing its viability. The halting problem itself is an undecidable decidability problem. The method used for proving these theorems uses the same strategy as the one used to prove the halting problem, which is to construct a Turing Machine which reaches a logical contradiction, either by determining the Halting Problem or otherwise. Many of these problems can be *reduced* to the Halting Problem, which is to say, it can be shown that it is logically equivalent to a subcase of the Halting Problem.

Undecidability of Runtime Bounds in P

There does not exist a Turing Machine which can determine the runtime complexity of another Turing Machine, given that it's in polynomial time.

Proof. Given Turing Machine M and an input tape x, construct new Turing Machine H which, given a description of M, x, and a separate string of length n as inputs, simulates the program of M on x for n steps. If M has halted after these n steps, H executes $\mathcal{O}\left(f\left(n\right)\right)$ arbitrary moves and halts. Otherwise, H executes $\mathcal{O}\left(g\left(n\right)\right)$ arbitrary moves and halts. Then, if there exists an n such that M halts on x in n steps, then H has a runtime of $f(n)$. If not, then for any string length n, H executes in runtime $g(n)$. If the runtime of H were decidable, it would imply the decidability of whether or not M halts on x, since M halting directly determines the runtime of H. However, it was shown that halting is undecidable, which leads to a contradiction. Therefore, there doesn't exist a Turing Machine that can determine runtime complexity, as desired.

Rice's Theorem

Rice's theorem states that no nontrivial semantic property of a given Turing Machine can be decidable. A semantic property of a Turing Machine is one that concerns its behavior, like whether or it halts on all inputs or its runtime. It can be seen as a more general version of the Halting Problem, but the proof relies on a reduction to the Halting Problem. A semantic property is *trivial* if it is true or false for all Turing Machines.

Proof. Assume there exists a Turing Machine Q which takes a description of a Turing Machine as input and decides whether it has some semantic property P. Let a Turing Machine A have the property P. A must exist from the non-triviality of P. Let B be another Turing Machine whose semantic properties are unknown. Construct a Turing Machine T which takes the description dB and a string j as input and simulates B on some predetermined input tape i. If B halts on i, it proceeds to simulate A with the input of j. If that process halts, T returns its output.

Consider the output of Q with an input of dT. First, we track what happens to T. If B halts on i, then in the following steps of computation T is functionally identical to A, as the program of A is simulated within T, and they produce the same outputs given the same inputs. Thus, whatever properties held by A are also held by T. Since A has property P, so does T, so Q outputs "yes". On the other hand, suppose B does not halt on i. Since T simulates B on i, T does not halt for any input string j. This means T has an undefined output at all points, and conventionally it does not have any semantic properties. Therefore, T does not have property P and Q returns "no".

This result implies the decidability of B halting with an input of i. B halting on i directly leads to Q outputting "yes" with an input of dT. Similarly, if B does not halt on i it directly leads to Q outputting "no". This implies Q can determine halting, which is known to be impossible. This is a contradiction. Therefore, no such Q can exist in the first place, as desired.

Computability of Numbers and Functions

Applications of the Halting Problem can be extended to many other problems involving numbers and functions, rather than just decideability problems. We say a number is *computable* if it can be calculated to arbitrary precision using a terminating algorithm. This is different from whether or not a number is irrational or transcendental. Well known constants such as π and e are irrational and transcendental, but they can still be calculated to arbitrary precision using techniques such as Taylor Series. However, most real numbers are uncomputable, as it can be shown that the number of computable numbers is countable due to a countably infinite number of possible Turing Machines and an uncountably infinite number of reals. Some examples of such numbers are Chaitin's Constant and the Busy Beaver Numbers. We present proofs on their uncomputability using reductions to the Halting Problem, similar to what was done previously with decideability problems.

Chaitin's Constant

Let p be a program, or tape, expressed in binary. Chaitin's Constant is the probability that some Turing Machine halts on p. Or, more formally, for a given Turing Machine U, we can define the Chaitin Constant as

$$\Omega_U = \sum_{U \text{ halts on } p} 2^{-|p|}$$

where $|p|$ is the length of p. Of course, this number varies by Turing Machine, and for some not even a single bit can be computed. We can show that if we know all bits of Chaitin's Constant, it's equivalent to solving the Halting Problem.

Proof. First, we show that knowing n bits of Ω_U for some Turing Machine solves the Halting Problem for programs up to n bits in size. For each $k \in \mathbb{N}$, we run all inputs of up to size k on the Turing Machine for k seconds. Some of these programs will halt in that time limit, and some won't. We keep a running halting probability Ω_k, which is the probability that a program of up to length k halts, which updates as k increases. Note that $\Omega_k < \Omega_U$ for all values of k, since we are only considering a subset of halting programs for any k. As k becomes larger, the space of programs that are tested approaches the universal set of programs, so Ω_k will approach the true halting probability Ω_U. That is to say, $\lim_{k\to\infty} \Omega_k = \Omega_U$. Ω_k becomes arbitrarily close to Ω_U, so for all n there exists a k such that the first n bits of Ω_k matches with Ω_U. At this point we will have encountered all programs of length n that halt, since any more programs of length less than n that halt will increase Ω_k such that $\Omega_k > \Omega_U$, which is impossible. As n ranges over the positive integers, we will have solved the Halting Problem. This is a contradiction, as the Halting Problem is undecidable. Thus, we cannot know all bits of Chaitin's Constant.

The Busy Beaver Numbers

Given a Turing Machine with n states and a tape alphabet of $\{0, 1\}$, what is the maximum number of 1's the machine prints on a blank tape before halting? Since there are a finite number of n-state Turing Machines, this number must exist and be well defined. We call this number $\Sigma(n)$, as a function of n. Surprisingly, this function grows faster than *any* computable function $f(n)$, and is therefore uncomputable. We show that this is the case; the proof is quite short.

Proof. Assume for contradiction that there exists some Turing Machine B that takes a number n as input and outputs $\Sigma(n)$. We may construct a Turing Machine M which takes a description of another Turing Machine dT as input, where dT encodes the number of states contained in T. Suppose T has k states, then M can determine that in finite time and pass the output to B. B computes $\Sigma(k)$ and passes it back to M. Using this, M simulates T on a blank tape for $\Sigma(k)$ steps. At the end of the simulation, T can have halted or is still going. $\Sigma(k)$ is the maximum number of steps that can occur for any Turing machine with k states, so if T hasn't halted, then it must never halt. However, this implies M can solve the Halting Problem for T in $\Sigma(k)$ steps, which is a contradiction. Therefore, B cannot exist.

Conclusion

We end on some problems.

1. Does there exist a Turing Machine which decides whether or not another Turing Machine prints a certain string?
2. Consider a Turing Machine with an *oracle* which can instantly solve the Halting Problem for normal Turing Machines. Can this Turing Machine solve the Halting Problem for other Turing Machines with oracles?

References

1. Dewdney, Alexander K. The (New) Turing Omnibus: 66 Excursions in Computer Science. Freeman, 2001.
2. j2kun. "Busy Beavers, and the Quest for Big Numbers." Math and Programming, 16 Feb. 2016, jeremykun.com/2012/02/08/busy-beavers-and-the-quest-for-big-numbers/.
3. Matos, Armando. "Direct Proofs of Rice's Theorem". 2014, https://www.dcc.fc.up.pt/ acm/ri-cep.pdf.
4. Chaitin, Gregory. "The Halting Probability Omega: Irreducible Complexity in Pure Mathematics". 23 Nov. 2006, arXiv:math/0611740.
5. Chaitin, Gregory. "The Limits of Reason". Scientific American. 294. 74-81, 2006, 10.1038/scientificamerican0306-74.
6. Keats, Jonathon. "How Alan Turing Found Machine Thinking in the Human Mind." New Scientist, 29 June 2016, www.newscientist.com/article/mg23130803-200-how-alan-turing-found-machine-thinking-in-the-human-mind/.