Computability: Turing Machines and the Halting Problem

Jeremy Booher

July 9, 2008

1 Effective Computability and Turing Machines

In Hilbert's address to the International Congress of Mathematicians, he posed the problem of devising a method to check whether a polynomial equation possessed any integral solutions. In 1900 there was no machinery to formalize what Hilbert meant by the phrase "To devise a process according to which it can be determined by a finite number of operations whether the equation is solvable in rational integers". And no one dreamed that it was provably impossible to do so. A formal definition of computation had to wait until the 1930s, when Alan Turing and his contemporaries investigated Turing machines, the λ-calculus, μ-recursive functions, and Post systems as models of computation. All of these methods of computation turn out to be equivalent. In addition, although working before the construction of the first modern computer, it is straightforward to write a computer program that will simulate an arbitrary Turing machine and to describe a Turing machine that will simulate a modern computer. Alonzo Church theorized that these methods of computations embody what we think of as "effectively computable". This is a hypothesis and not a rigorous mathematical statement so it cannot be proven, but was widely accepted after Turing's work. Known as the Church-Turing thesis, it states "Every function which would 'naturally be regarded as computable' can be computed by a Turing machine." In practice, we therefore rarely describe a Turing machine formally, but instead say something informally such as "Given an input $< M, x >$, where M is a TM and x is the input, run M on input x, and accept if it accepts or rejects". A formal definition of a Turing machine is in the appendix. This abstraction away from the rigorous definition is analogous to the way that people program in high level languages like Scheme or C and ignore assembly language and the details of computer architecture.

2 Decision Problems

The simplest class of problems to consider are those which, given an input, require a yes or no answer. For example, given an integer, return yes if it prime and no otherwise. These are known as decision problems. Formally, given an input alphabet Σ and a subset A of Σ^* (Σ^* is the set of finite strings formed by concatenating elements of Σ along with the string "", and is called the Kleene-star of Σ), we want to determine whether $x \in \Sigma^*$ is in A. To do so, we want a Turing machine M that will accept x iff $x \in A$. The set $\{x \in \Sigma^* : M \text{ accepts } x\}$ is called the language of M, and denoted by $L(M)$. We call a language A recognizable (recursively enumerable) if there is a Turing machine M with $L(M) = A$. We call a language A co-recognizable (co-r.e.) if $\Sigma^* - A$ is recognizable. If a language is both recognizable and co-recognizable, it is called decidable (recursive). In other words, a recognizable language is one where we can always check if a string is in it, but we cannot tell if a string is not in it (because the Turing machine M may not halt), a co-recognizable language has a Turing machine that will halt and accept if the input is not in the set and may reject or never halt if it is in the set, and a decidable language is one that has a Turing machine that will always halt and give a yes or no answer as to whether its input is in the language.

The language $\{n \in N : n \text{ is prime}\}$ is decidable since a program that checks whether any integer from 2 to $\sqrt{n}$ is a factor can tell whether or not n is prime. Here's another example of a decidable language:

$$\{< M >: M \text{ (a Turing machine) halts on the empty string within 55 steps } \}$$

Here, we use the notation $< M >$ to denote some representation of the machine M encoded using the symbols of Σ. The exact representation doesn't matter, only that such a representation exists. Since every TM encodes only a finite amount of information (any computer program is finite) there is no problem in describing a Turing machine in terms of a finite string. There are numerous possible encodings - the most familiar is certainly that on a computer: all programs are stored in memory as a sequence of bits, so there is a natural correspondence between programs and natural numbers. Given some sort of representation, we let M_x denote the TM that corresponds to the string x.

Additionally, there is a universal Turing machine U that given input $< M, x >$ (a description of a TM and input x) will essentially run M on x : accept if M accepts x, reject if M rejects x, never halt if M never halts on x. Turing gave an explicit construction of a universal machine and encoding in his papers, but in light of the Church-Turing thesis we know this is possible because there are virtual machines for languages like Java and Python that are computer programs that simulate a computer.

Now that we know about representations and the universal Turing machine, let's return to

$$L = \{< M >: M \text{ (a Turing machine) halts on the empty string within 55 steps } \}$$

Let N be a TM that on input $< M >$ runs M on the empty string (we can do this since we have a universal machine). We can modify the universal machine so it counts how many state changes M makes, and have it stop after 55 steps. If M halts of its own accord, accept, otherwise reject after the 55^{th} step. N always halts and $L(N) = L$ making L decidable.

3 The Halting Problem

It would very nice if we could tell whether running a TM M on input x will halt, since if we could do this, any recognizable language would also be decidable. We could then construct a machine that searches for counterexamples to an unsolved problem, and check whether it halts (finds one) in order to see if the unsolved problem is true. So how would we go about checking?

It is easy to see that some Turing machines never halt. Consider the machine that starts at 1 and adds one repeatedly. When it gets to 0, it accepts. Although we know this machine never ends, Turing showed that figuring this out for arbitrary Turing machines is impossible.

Theorem 1. *The language $\{< M, x >: M$ halts on input $x\}$ is recognizable but not decidable.*

Proof. This set is obviously recognizable: simply run M on x (using the universal machine) and accept if M halts. Now, suppose for the sake of contradiction that we had a machine H which decided the halting problem. In other words, H takes $< M, x >$ and accepts if M halts on x, and rejects if M does not halt. Let the machine N work as follows: it takes its input x and writes $< M_x, x >$ on the tape (remember M_x is the TM whose representation is x). Then it runs H (using the universal TM) on $< M_x, x >$, and accepts if H rejects, and enters an infinite loop if H accepts.

Now, N has a description y. Consider what N does on input y. Well, by our construction it simply runs H on $< M_y, y >$, which checks what N does on y. If H accepts, then by our construction N loops. However, H accepting on $< N, y >$ meant that N halted, contradicting the fact that N looped. Likewise, if H rejects, this means that N accepts (and halts), which contradicts the fact that H rejected, meaning N didn't halt. Therefore, we have a contradiction in either case, so there cannot exist a decider for the halting problem. □

Corollary 2. *The halting problem is not in co-recognizable. In other words, no Turing machine can recognize all Turing machines that never halt.*

Proof. The halting problem is recognizable but not decidable. The set of all languages that are recognizable and co-recognizable are the decidable languages, so if the halting problem were co-recognizable then it would be decidable. □

4 Other Undecidable Problems

Now that we know the halting problem is not decidable, we can use this to show that many other problems are not decidable. A general way to do this is to assume that we had a decider for the problem, and then use it to construct a decider for the halting problem. Since the halting problem is not decidable, we have a contradiction, and the first problem is therefore not decidable.

Consider the finiteness problem. Let

$$FIN = \{< M >: L(M) \text{ is finite } \}$$

We will show this is not recognizable by reducing it to $\overline{HP}$. (This is the negation of the halting problem. $\overline{HP} = \{< M, x >: M \text{ does not halt on } x\}$) Obviously, $\overline{HP}$ is co-recognizable but not recognizable. Now, let's assume there is a TM F that accepts input $< M >$ if $L(M)$ is finite (and may reject or not halt if $L(M)$ is infinite). How can we recognize $\overline{HP}$ using F? Given input $< M, x >$ (a TM and an input x to check if it halts on), construct a TM N that does the following:

1. On input y, erase y from the tape.
2. Write x on the input tape
3. Run M on x.
4. Accept if M halts.

Now, to check if a TM M halts on x, we just need to construct a description of this machine N, and run F on N. If F accepts, then $L(N)$ is finite. But since N ignores its input, $L(N)$ is finite means $L(N) = \emptyset$, so M does not halt on x. If F rejects, then M does halt on x. Therefore, F recognizes $\overline{HP}$, which contradicts the fact that $\overline{HP}$ is not recognizable. Therefore, FIN is not recognizable. (One subtle point: is it possible to construct a description of N based just on $< M, x >$? Yes it is, and it should be obvious this is possible since compilers are computer programs that produce other computer programs.)

It turns out FIN is not co-recognizable. This can be shown by deciding the halting problem with a recognizer for $\overline{FIN}$. Given an instance of the halting problem $< M, x >$, produce a machine N that:

1. On input y, save y on some unused section of the TM tape, and write x in the input position.
2. Run M on x for $|y|$ (length of y) steps.
3. Accept if M has not halted. Reject if M has halted by this time.

If M does not halt on x, then N accepts y, which means $L(N) = \Sigma^*$, so $< N >$ is not in FIN. If M halts on x in t steps, N accepts inputs of length less than t and rejects longer ones, which means that $L(N)$ is finite. Therefore, $< N >$ is in FIN iff M accepts x, so we can tell when a machine does not halt with a recognizer for FIN. Therefore, since $\overline{HP}$ is not recognizable, FIN is not co-recognizable. But we already showed that FIN is not recognizable, which means no Turing machine can always determine whether a TM N accepts a finite number or an infinite number of strings.

Another undecidable problem that does not deal directly with the behavior of Turing machines is the Post Correspondence problem. Consider the set of "dominos"

$$\{\frac{b}{ca}, \frac{a}{ab}, \frac{ca}{a}, \frac{abc}{c}\}$$

The goal is to place them in a linear order (with repeats) so that the string on the top reads the same as the string on the bottom. Each domino may be used arbitrarily many times. For example, a solution to the sample problem is

$$\frac{a}{ab} \; \frac{b}{ca} \; \frac{ca}{a} \; \frac{a}{ab} \; \frac{abc}{c}$$

However, in general solving this problem is undecidable. By clever encodings, it is possible to represent the computational history of a Turing machine (what it did at each time step of its computation) as strings in the Post correspondence problem so that they can fit together in a solution iff they Turing machine has an accepting computation. Therefore, being able to solve the Post correspondence problem (or even detect if there is a solution) would decide the halting problem. As opposed to halting problem, this problem seems more natural since it does not directly reference Turing machines.

Finally, consider Hilbert's 10^{th} problem, to devise an algorithm to determine whether a given Diophantine equation has any solutions. Yuri Matijasevic showed in 1970 that this is undecidable too. Although specific cases like linear Diophantine equations are easily solvable, there is no universal method to determine the solvability of Diophantine equations.

5 Rice's Theorem

We've seen two examples of problems about Turing machines that are undecidable. In fact, the only questions about Turing machines that we've been able to answer so far are trivial ones such as whether a given machine halts in a fixed number of steps or has a certain number of states. This is no coincidence.

Definition 3. *A property is a map P from the set of TM to $\{0,1\}$ (false/true) such that $L(M) = L(M')$ implies $P(M) = P(M')$. If $L(M) = A$, the language A has the property if $P(M) = 1$. A property is nontrivial if there exist TM M and M' such that $P(M) = 1$ and $P(M') = 0$.*

This rather arcane definition simply says that properties must depend on the language a machine describes, not the specifications of the machine. For example, a computation taking at most 55 steps is rather meaningless, since I can write a machine for an identical language by modifying the machine to first calculate 10 digits of π and then solving the problem. The quality "takes at most 55 steps" is not a property since it doesn't say anything about the language, only the machine.

Theorem 4. *If P is a nontrivial property, then the problem "Does $L(M)$ have property P" is undecidable.*

Proof. Since P is nontrivial, we may as well assume that $\emptyset$ does not have the property and there is a language A that has the property. Let K be the TM with $L(K) = A$. The language under consideration is $P^{-1}(1)$ (the set of Turing machines M for which $P(M) = 1$). We'll reduce the halting problem to this problem. Given $< M, x >$, produce a TM N that on input y :

1. Saves y onto some unused section of the TM tape.
2. Writes x on the tape.
3. Runs M on x.
4. If M halts on x, run K on y: accept if it accepts.

If M halts on x, then N accepts if K accepts y, which means $y \in A$ and $L(N) = A$. If M does not halt, then y is not accepted, so $L(N) = \emptyset$. Therefore, if M halts on x, $L(N) \in P$, and if M does not halt, $L(N) \notin P$. Therefore, a decider for the language $L = \{M : P(M) = 1\}$ would decide the halting problem. Since HP is not decidable, the language L is undecidable. □

6 The Arithmetic Hierarchy

Now that there are so many undecidable problems, it would be nice to have a measure of "how hard" they actually are. For example, FIN should be harder than HP, since we can at least recognize the language of halting Turing machines, while we cannot recognize the finite or infinite languages. However, there are many languages that are neither recognizable nor co-recognizable, and we still need some way to compare them. This leads to the notion of an oracle.

Definition 5. *An oracular Turing machine is a normal Turing machine with access to a yes/no oracle for a language L. At any step, the Turing machine may query the oracle with a string x and immediately learns whether $x \in L$.*

The most common oracle to study is an oracle for the halting problem. Denote this oracle by HP_0. With this oracle, the halting problem is decidable, since a machine, given an instance of the halting problem, simply returns whatever answer the oracle gives it. Likewise, the language $\{< M, x >: M$ accepts $x\}$ is decidable relative to this oracle.

Definition 6. *Inductively define HP_0 to be an oracle for the halting problem, and HP_n to be an oracle for the language $\{< M, x >: M$halts on $x\}$ where M is a Turing machine with access to the oracle HP_{n-1}. Let Σ_n be languages recognizable by oracular Turing machines with access to the oracle HP_{n-1}, Π_n languages co-recognizable by these Turing machines, and let $\Delta_n = \Sigma_n \cap \Pi_n$. Let Σ_0 be the recognizable languages for Turing machines with access to no oracle, and Π_0 the co-recognizable ones. Δ_0 is the decidable languages.*

For example, we know $HP \in \Sigma_0$ but not in Π_0. It is obviously in Δ_n for $n \geq 1$. In addition, $\Sigma_n, \Pi_n \subset \Delta_{n+1}$. The collection of all these classes is called the Arithmetic Hierarchy. There are plenty of technical details to check to make sure this doesn't collapse and is well-defined[1]. The most interesting result is the following theorem, due to Post.

Theorem 7. *Σ_n is the languages of the form*

$$\exists x_1 \forall x_2 \exists x_3 \ldots Q_n x_n P(x_1, x_2, \ldots, x_n)$$

Π_n is of the form

$$\forall x_1 \exists x_2 \ldots Q_{n-1} x_n P(x_1, x_2, \ldots, x_n)$$

where P is any formula of first order logic and Q_n is $\forall$ if n is even, $\exists$ otherwise.

Although there are some details to check, the proof of the theorem boils down to the following idea. To check a statement of the first form, the Turing machine can make an exhaustive search of all x_1. Then, for each x_1, it uses the oracle HP_{n-1} to check whether an exhaustive search through the $x_2 \dots x_n$ to determine whether $\forall x_2 \exists x_3 \dots Q_n x_n P(x_1, x_2, \dots, x_n)$ ever terminates. (To do this formally, use induction to show this subproblem will be in Π_{n-1}.) As the machine searches through the x_1, if it ever finds an x_1 that makes the sub-formula true, it returns true. Clearly then, this language is in Σ_n. A similar idea works for Π_n.

This gives a proof that FIN is not that hard after all.

$$FIN = \{M : \exists t \forall x, |x| > t \rightarrow M \text{ does not accept } x\}$$

so it is in Σ_1. It can be shown that $FIN \notin \Pi_1$, so the smallest class containing FIN is Σ_1. In contrast, there are other natural problems which are outside of Δ_2.

7 And now

This is just a taste of computability theory. There is a whole theory about models of computation with less resources than TM, such as finite automata, context-free grammars, and push-down automata. These are incapable of recognizing many common languages, such as

$$\{a^n b^n c^n : n \geq 0\}$$

Sipser's book[3] is an excellent reference on this topic and most others. For more information on the arithmetic hierarchy, see [2] or [1]. Sipser also contains a longer discussion of the Post correspondence problem.

8 Appendix

8.1 A Formal Definition of a Turing Machine

A Turing machine has an infinite (in one direction) tape, with one cell highlighted as the current cell. The machine also keeps track of its current state, and it has a book of rules for determining what to do based on its current state and what is written in the current cell. Every rule tells the machine to do three things: change the current state, write a symbol in the current tape cell, and then move the current cell left or right. To begin a computation, the input is written on the tape (starting in the first cell) and the machine begins in the first cell and in the start state. In addition, the machine has two special states, the accept or reject states, and entering one of them signals that the machine has finished and gives a yes or no answer. More formally,

Definition 8. *A deterministic, one tape Turing machine is a 9-tuple*

$$M = (Q, \Sigma, \Gamma, \diamond, \_, \delta, s, t, r)$$

where Q is a finite set of states, Σ is a finite set whose elements are the input alphabet, and Γ is a finite set containing Σ whose elements are the tape alphabet. $\_ \in \Gamma - \Sigma$ is the blank symbol, $\diamond \in \Gamma - \Sigma$ is the endmarker, $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function, $s \in Q$ is the start state, $t \in Q$ is the accept state, and $r \in Q$ is the reject state. (Note: $\delta(p, a) = (q, b, d)$ means when in state p with symbol a on the current tape cell, change to state q, write b on the tape, and move in direction d (left or right).) For simplicity, let's restrict δ so that the endmarker $\diamond$ is never overwritten or moved beyond (so that we don't have to specify what the machine does if it attempts to move off the tape). Furthermore, we require that M never leave the accept or reject state (so that we can talk about a computation finishing when it accepts or rejects).

Now, M runs on x (x is the input, a string of letters in the input alphabet Σ) as follows: first $\diamond$ is put in the first cell of the tape to mark the tape end, then x is written on the tape, one letter in each cell, and finally the rest of the tape is filled with the blank symbol $\_$. The current cell is set to be the first (containing $\diamond$), and the current state is s. Then the Turing machine runs by repeatedly calculating $\delta(p, b)$, where p is the current state and b is the symbol in the current cell. This is a 3-tuple (q, b, d), and the current state becomes q, the letter b is written in the current cell, and then the current cell moves either left or right (d).

Here's an example of a Turing machine. (This description could be made completely rigorous using the 9-tuple above, but in practice this is never done since it's nearly impossible to understand and painful to produce.) Our goal is to accept or reject (give a yes or no answer) to the question of is a sequence of a's, b's, and c's of the form $a^n b^n c^n$ (n a's followed by n b's followed by n c's). For example, "aabbcc" should be accepted, as should "abc" and "" (the empty string), while "abbc", *bac*, and the like should be rejected. Remember that the machine starts with the input (x) written on the tape and the head at the endmarker. We want the machine to scan from left to right, and check that all the a's come before the b's and the b's come before the c's. It is easy to do this using separate states for "seen an a" "seen a b" and "seen a c". If this condition is not satisfied, the machine should enter the reject state. When the machine sees an empty cell (the end of the input), it should write a special symbol (say #) there. Next, it successively sweeps left and right between the $\diamond$ and #, and at each pass it replaces the first a, the first b, and the first c it sees with a blank ($\_$). If it does not see one or two of the letters on a pass between the $\diamond$ and #, then it should reject, and it should accept if it saw no letters at all. Thus, this machine accepts if the input was of the form $a^n b^n c^n$, and rejects otherwise. In particular note that this machine will always halt, since on each pass after the first it crosses out an a, and since the input is of finite length there are only a finite number of a's.

8.2 Alternate Types of Turing Machines

Our definition of Turing machines has many possible generalizations. For example, instead of allowing our tape to be unbounded only in one direction, why don't we let it extend infinitely in both directions? However, this alteration will not effect what the Turing machine can do, since we can simulate this "two way" Turing machine on a normal Turing machine. We'll let cell $2n$ of the standard TM represent the n^{th} cell to the right of some fixed cell on the two-way tape, and cell $2n + 1$ represent the n^{th} cell to the left of the fixed cell. Through rewriting the transition function δ, we can convert a two way Turing machine into a normal Turing machine. This will sometimes change the time complexity of a problem, but it will not effect what Turing machines can do.

Another extension could be to make the TM nondeterministic, which means that instead of having just a single state and a single tape, at every step it can "branch" and make multiple choices. For example, in state x with a written in the current cell, it might go either left or right. All of the branches are executed at the same time, and if any branch accepts or rejects, the entire machine accepts or rejects. However, this too is computationally equivalently to a normal Turing machine (although it certainly looks like it will work faster than a normal TM). To convert a NTM to a TM, we just need to use something like parallel processing (or multi-threading) where we execute steps of each of the branches in sequence. Therefore, nondeterministic TM have the same computing power as regular TM.

References

[1] Martin D. Davis and Elaine J. Weyuker, *Computability, complexity, and languages*, Academic Press, Inc., 1983.

[2] Dexter Kozen, *Automata and computability*, Birkhauser, 1997.

[3] Michael Sipser, *Introduction to the theory of computation*, Thompson Course Technology, 2006.