Computability: Extremely Short Guide

Vitaly Parnas

March 09, 2018

I summarize the typical content in the Theory of Computation[1], omitting detailed proofs and focusing on the higher level concepts. This area helps develop intuition in determining what sorts of problems are theoretically possible to address on a modern computational unit and what problems cannot be computed.

Helpful prerequisites: Asymptotic notation, set-builder notation.

Language building blocks and operations

A *language*, not to be confused with a more common association of a programming language, is a fundamental entity in the theory of computation that can represent a set of inputs valid in a certain domain (equivalent to a valid input in a modern computer program), or a more abstract set of strings conforming to certain rules, such as {banana, tractor} or a set of all even numbers. A *string*, accordingly, consists of *symbols* from a certain *alphabet* such as $\{a, b, 0, 1\}$.

Languages can undergo operations such as a *concatenation* (a combination of all strings from multiple languages), *kleene star* (*) and *plus* (+) operations familiar from regular expressions, as well as set theory *union*, *intersection*, and *complement* operations.

Countability

A crucial measure in leading to determine what's computable is *countability*. Any set is *countable* if it maps to natural numbers, or more intuitively, if you can simply count each element. You could argue that any set is countable - you simply count the elements: 1, 2, 3... However, the matter complicates once we introduce sets of other sets.

A countable union of countable (possibly infinite) sets is still countable, although we must count the sets in a particular diagonal fashion. If you were to simply count each set horizontally - sequentially, you would never reach the end of an infinite set before being able to proceed to the next. Similarly, if you counted vertically, in presence of an infinite number of sets, you would never proceed past the first element of any individual set.

A set of languages over a finite alphabet, however, is *uncountable*. Let's imagine a set of *all* infinite binary strings, a simple case. The proof involves the *diagonalization* trick, which leads to a paradox: if we vertically stack different infinite binary strings, at some point the set must include a valid language in, let's say, position k, $L_k = \{x_i | x_i \notin L_i\}$, that is, L_k must be different from every preceding row in the table, and effectively the bit in the ith position becomes the opposite of the corresponding bit in language L_i. Consequently, the kth bit in language L_k must be the opposite of itself, which leads to a paradox. This may seem strange, but intuitively, a set leading to elements that you cannot express, is uncountable.

A powerful corollary to the above is that a set of all binary functions is uncountable and hence exist some functions we cannot write programs for or even conceptualize.

Turing Machines

A *Turing Machine* is an abstract model of any computational unit. Anything that is computable in a modern sense, can, theoretically (albeit impractically), be modelled by a TM. A TM consists of an infinite tape of cells, the current tape position, the tape alphabet (ex: symbols 0 and 1), the tape contents, a set of states (such as in a finite state machine), a transition function mapping the current state and cell content in the present tape position to a new state, new (or unchanged) cell content, and movement left or right along the tape. A TM also includes an initial state, an accept state, and a reject state.

A language of all even numbers, for example, is *decided* (computed) by a TM if it reaches an accept state upon any valid input (an even number), and a reject state upon anything that's not an even number. The TM cannot loop forever if it's considered a decider - it must eventually accept or reject all inputs. A softer variant, a *recognizer*, is a TM that must simply accept any valid input, and not accept an invalid input, although it could indefinitely loop.

Machine Equivalence

Anything that's computable by a modern CPU is computable by a TM and vice-versa. This includes other, more elaborate TM models. For example, one such model is a *Stay Put* TM, which allows the tape head to stay put after a transition instead of proceeding left or right. Another model, frequently handy in demonstrating a language complexity class or equivalence to other models is a *multitape* TM, with a more elaborate transition function that can transition and modify each tape independently upon each step. In addition, a *RAM* model resembles a more traditional CPU, containing registers, a program counter, a program consisting of a series of possible instructions, and a single-instruction read/write memory. The RAM model can be simulated by a multitape TM with different tapes corresponding to the individual RAM model components.

In general, all of these models are equivalent in the sense that one can simulate another irrespective of the increase or decrease in model complexity, as long as the simulation complexity is limited by a polynomial factor - that is, the simulation of one model by another is of $O(n^k)$ complexity and does not incur an exponential cost (with respect to input size). A single-tape TM can simulate a multitape TM, for example, by means of an expanded alphabet containing hash symbols and dots, allowing keeping multi-tape contents on one tape via separators and using dots to mark individual head positions.

Universality

A TM can receive another TM encoded as input, simulate the encoded TM, and accept/reject as the encoded TM would. In fact, this is representative of what happens in a modern computer that stores different programs on disk, loads them into memory on demand, and executes them. A 3-tape TM can respectively receive an encoding of another TM on one tape as input, use the second tape to store and analyze the transitions, and the third tape to track current state.

Recognizability/Decidability

A language is considered *recognizable* if exists some TM that can recognize it (accept if the input is valid for the language). Similarly, a language is *decidable* if some TM can decide the language - not only accept valid input but reject the invalid. Intuitively, if a language and it's complement are *recognizable*, the language is *decidable*. Demonstrating this fact requires running two TMs for each, incrementally, and in parallel, since passing invalid input to a machine recognizing a certain language can cause it to loop forever if the machine doesn't reject, while not giving a chance to the second machine that *would* recognize the complement, accepting this same input as valid.

Here's an example of an undecidable language: $L = \{< M > | M \text{ doesn't accept } < M >\}$. This is a language that takes a machine M encoded as input, and accepts if and only if M doesn't ac-

cept the encoded version of itself. Demonstrating undecidability of L involves a similar diagonalization trick as the uncountability of a set of languages or infinite binary strings. It results in a paradox.

Mapping Reductions

A *mapping reduction* is a way to demonstrate *decidability* or *undecidability* of a language with respect to another language of which this factor is already known. Language A is *mapping reducible* to B if there exists a computable function f such that an input w is valid in A if and only if $f(w)$ is valid in B. In other words, the entire set of valid input in A is mapped to some valid input(s) in B, and not necessarily in one-to-one fashion. Providing such a mapping reduction for a language demonstrates that B is at least as hard as A. As a consequence, knowing A is undecidable implies that B is undecidable, and knowing B is decidable implies that A is decidable. Such proofs generally involve only that: constructing a Turing Machine that computes a mapping reduction from one language to another, the decidability of one of which we already know, which unravels the decidability or undecidability of the other by one of such implications.

Halting Problem

The *Halting Problem*, $H_{TM} = \{< M, w > | M$ halts on $w\}$, is a language that receives a TM M and a string w encoded as input, and accepts if M halts (accepts and rejects) appropriately on w. This is an undecidable problem, proven by a mapping reduction from a number of other undecidable languages. The important implication of the halting problem involves infinite loops. We cannot reliably check a modern program for the existence of an infinite loop. We can certainly detect if a program terminates, in respect to which, H_{TM} is actually recognizable, since we can build a machine to accept upon a valid input. The converse, however, is not practically detectable. Intuitively, there is no way to definitively determine if a supposedly looping machine will ultimately terminate.

References

[1] Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, Boston, MA, third edition, 2013.