Universality in the theory of algorithms and computer science

Alexander Shen

Computational models

The notion of "computable function" was introduced in 1930ies. Simplifying (a rather interesting and puzzling) history, one could say that several researchers (Turing, Post, Kleene, Church and others) have defined a class of functions, now called "computable functions". A function f (whose arguments and values are constructive objects, say, integers or strings) is called *computable* if there exists a machine M such that

- $f(x)$ is defined $\Rightarrow$ M terminates on input x with output $f(x)$;
- $f(x)$ is undefined $\Rightarrow$ M never terminates on input x.

One should also specify the class of "machines" used in this definition. Turing and Post defined rather similar classes now called "Turing machines" and "Post machines". Other definitions were more esoteric: Church and Kleene defined "partial recursive functions" as a smallest class of functions that contains basic functions (projection, identity, successor) and is closed under some algorithmic constructions (substitution, recursion and μ-operator). Later other classes of machines were introduced (Markov's "normal algorithms" based on string search/replacement operations, register machines that have finite (or countable) number of integer registers and allow programs with assignments, **if-then-else-fi** and **while-do-od** constructions, and many other classes).

This development poses many interesting historical and philosophical questions. One of them: Why at the same time many different researchers independently came to the same notion of computability? (This notion was not inspired by practice, since at that time no computers exist. The definition does not use any mathematics not known in XIX century.)

But now we want to stress the following universality phenomenon: if we consider any reasonable computational model (class of abstract machines), we come to the same class of computable functions.

Of course, if the model is extremely restrictive, it is not the case. For example, if a Turing machine is allowed only to read the tape symbols but is unable to change them, this machine can recognize only few sets (machine essentially becomes a finite automaton). Or if a program for register machine uses only one register, it can not do much. But if we weaken our restrictions allowing more and more freedom, at some point our model becomes universal and further changes are not needed. For example, a register machine with 3 registers can compute the same functions as the machine with infinite number of registers.

It is like counting "one, two, many" — at some point we fall into the universality trap and further changes become inessential. Another miracle is that this trap is the same for many completely different machine architectures. E.g., register machines and Turing machines define the same class of computable functions. The famous Church–Kleene-Turing thesis says that this class of functions corresponds to an intuitive notion of "effectively calculable" function.

The "universality" observation can be formulated as follows: *if a computational model is not too primitive, it is universal* (can simulate any other computational model).

Circuits

Another, much simpler, universality phenomenon can be observed with Boolean circuits. Consider gates that have inputs (one or many) and output; output voltage can be "high" or "low" depending on input voltages (which also can be high or low). Identifying "high" with 1 and "low" with 0, we say that gate computes a Boolean function from $\{0,1\}^n$ to $\{0,1\}$ (where n is the number of inputs). Several gates could be combined into a circuit by connecting output of some gates to inputs of other gates (loops are not allowed). Such a circuit computes a Boolean function of type $\{0,1\}^n \to \{0,1\}^m$, where n is the number of inputs and m is the number of outputs.

One could a priori expect that our ability to construct circuit for a given Boolean function depends on the set of available gates (the more complicated function we want to construct, the more diverse gates we need to use). At some extent it is true: for example, if we have only AND and OR gates, we can compute only monotone functions. But again, *if our gates are not too special* (not all monotone, not all linear, not all self-dual, not all conserving 0 and not all conserving 1, whatever it means), *the set of gates is universal* (any Boolean function can be computed by a circuit made of these gates).

Universal machine and stored program

The definition of computability says that for each computable function there exists a Turing machine that computes it. One may expect that for more and more complex functions we need more and more powerful machines (in terms of number of states or alphabet size). It turns out, however, that if we allow to prepend the input by some additional string ("the description of the machine simulated") that it is not the case: any machine can be simulated by some fixed "universal" Turing machine.

In mathematics this corresponds to the following theorem: there exists a computable function U (arguments and values are, say, binary strings) such that for any computable function f there exists a string p such that $U(px) = f(x)$ for any x (the equality means that either both $U(px)$ and $f(x)$ are undefined or both are defined and equal).

This theorem provides a base for the most important results in the foundations of mathematics (Gödel incompleteness theorem, the existence of effectively unsolvable problems).

In practice this construction is probably the most important invention of XXth century that allows thousands of appliances to share the same processing engine (and differ only by programs

executed by this engine); without this invention modern electronic industry would be impossible.

Stretching this observation a bit, we may claim that any sufficiently complex electronic device or program is universal (may be used to simulate any other machine). This is not literally true, of course, if a program or device is too specialized (even if it is very complex). But if the device is tunable enough, usually one can somehow find a "security hole" that allows to simulate any computation on this device.

Feasible computations

The notion of computability does not restrict the computational resources need to perform the computation. Some computable functions require huge number of steps and hence hardly can be computed in any practical sense. So it is natural to look for a smaller class of functions that reflects the intuitive idea of "feasible" computation that can be done with moderate resources.

Of course, we cannot hope to capture the idea of "feasibility" completely; a mathematical definition unavoidably would be of some asymptotic nature (it would be strange to say that computation with N steps is feasible while computation with $N+1$ steps is not). Most popular approach says that feasible computations are computations performed in polynomial time (i.e., there exists a polynomial p such that computation requires at most $p(n)$ steps for inputs of size n).

The interesting fact here is that this definition is robust: the same class is obtained for (almost) any reasonable model of computation (e.g., Turing machines with one tape, many tapes, cellular automata, register machines with registers of polynomial size, etc.)

So we have here the same kind of universality as we had for Church – Kleene – Turing thesis; one can state a "polynomial-time Turing thesis" that claims and any computation that can be performed by a reasonable device in polynomial time, can be performed by a Turing machine in polynomial time.

NP-completeness

One of the most important discoveries in the computational complexity theory is the discovery of the class NP and the notion of NP-completeness.

Roughly speaking, NP is the class of "search problems" where we need to find a solution among exponentially large search space, but each candidate could be easily checked.

There are many problems of this type. For example, we may look for a solution of a system of equations or inequalities. It is easy to check whether a given tuple of values satisfies the required condition. On the other hand, there are exponentially many tuples to be checked.

Another well-known example: a graph is given; we want to find a Hamiltonian cycle (that visits each vertex exactly once) or an Euler cycle (that traverses each edge exactly once). (It is easy to check whether a given sequence of edges forms a Hamiltonian or Euler cycle, but there are exponentially many sequences to test).

One more example (also about graphs): we want to know whether vertices of a given graph could be colored using k colors in such a way that no edge connects vertices of the same color.

(Each coloring is easy to test, but there are exponentially many colorings.) This problem is called k-coloring problem.

Problems of this type are called "NP-problems". For some NP-problems one can find a polynomial algorithm. For example, it is the case for Euler path problem (graph has an Euler path iff it is connected and each vertex has even number of adjacent edges; both conditions could be checked in polynomial time) or for 2-coloring. For other NP-problems nobody could either find a polynomial algorithm or prove that it does not exist.

It turned out (quite unexpectedly) that many of NP-problems for which polynomial-time algorithm is not found yet, are equivalent to each other in the following sense: if a polynomial-time algorithm exists for one of them, it exists for all others (and, moreover, for all NP-problems). These problems are NP-complete (an NP-problem is called NP-complete if any NP-problem is reducible to it). Now all NP-problems are divided into three classes:

• problems known to be solvable in polynomial time;

• problems known to be NP-complete;

• problems for which no polynomial-time algorithms is known and no proof of NP-completeness is known.

There are several important problems of the last type, e.g., the graph isomorphism problem (two graphs are given; one should find out whether they are isomorphic or not). Integer factorization also can be considered as problem of this type. But nevertheless most natural NP-problems turn out to be either polynomial-time solvable or NP-complete.

One can say that "if a NP-problems appears and we cannot find a polynomial-time algorithm for it, there are good chances that it is NP-complete". In other words, hard problems again tend to be universal.

Physics and computations

Looking for the justification of Turing thesis, we may try to reduce it to physics laws. Imagine that we have a physical system that has output (i.e., electric wire). Each second we measure the output value (voltage) which is either 0 or 1 (=low/high) and get a sequence of zeros and ones. Do the laws of physics guarantee that this sequence would be a computable one? If yes, could this result (or some related result dealing with input–output transformations) be considered as a derivation of Turing thesis from physics laws?

This question is rather difficult to discuss, since it is not well posed. First of all, the "laws of physics" are not an established consistent axiomatic system (as mathematicians would like to see them). They are more like a conglomerate of different notions, beliefs and recipes that (though being sometimes rather vague and even inconsistent) are very useful for the orientation in the real world and for inventing funny things (and not so funny things, like atomic bombs). Therefore the question whether something could be derived from laws of physics or not is not well posed. For example, sometimes physicists say that our world is finite in space-time, so only finite sequence could be generated, and all finite sequences are computable. Does this argument constitute a "fair" derivation of Turing thesis from physics laws (if we accept the finiteness of our world)? Probably not.

[Side remark: one should note also that Turing thesis does not say anything about human (or machine's) capabilities in general, it says only that any *algorithm* is equivalent to a Turing machine. If a human person can compute some non-computable function but cannot explain to anybody else how to do this, Turing thesis is not violated.]

To get a more reasonable question to physicists, one may restrict some resources available to a computational device. Imagine that we have a sequence ω of zeros and ones and a physical device that fits into cubic box of size $1\,m^3$, uses at most 1 kW of electric power, works at room temperature and, being switched on, produces a bit each second in such a way that a probability of event "nth bit equals ω_n" is at least 99% for each n.

Could we conclude from this information that sequence ω is computable and get some bounds on the computational complexity of ω? (This question seems to be related to another one, often discussed, but rarely formulated clearly: is energy dissipation inherently needed for a computation?)

Another approach: we can ask how difficult is to simulate some analog device with a digital one. Before the era of digital computers, some analog computers were used, e.g., for solving differential equations. However, after the appearance of digital computers the analog ones become obsolete: it turned out that the same things could be done with digital simulation (often with better precision and even faster). Later, following this trend, the analog processing of sound, images etc. was in many cases successfully replaced by digital processing. One may generalize this tendency and say that "analog devices could be simulated by feasible computations on digital computers", or even "any problem decidable with analog computers is polynomial".

Could we somehow "prove" this thesis by analyzing the laws of physics? One can say that the laws of mechanics correspond to ordinary differential equations which are not too hard to solve numerically. It is not so clear with partial differential equations (though sometimes they are just equation for averages of individual particle motions, like in hydrodynamics). Moreover, even if we agree in principle with this argument, the constants (that could be proportional to the number of particles in the system) may make this simulation practically impossible.

Things become even worse when we go to electrodynamics and quantum mechanics. The most serious obstacle is related to quantum mechanics. The system of n particles is represented there by a space which is a tensor product of the spaces representing individual particles. The dimension of this tensor product is exponential in n, so we need to compute eigenvalues of matrices of exponential size.

So one could expect that a problem of simulation laws of quantum mechanics (=predicting the behavior of quantum system, even in terms of probabilities) is hard.

Quantum computers

The same observation could be stated in a positive way: one can construct a quantum system that can compute something which is hard to compute by a classical system.

Most famous example of this type is Shor's algorithm for factorization: it describes some (imaginary) quantum system whose behavior can be used to factor large integers (assuming that this behavior is described completely by laws of quantum mechanics and noise can be ignored).

This is rather far from constructing a practical device for factorization, since it is not known how to create a real system that has necessary structure and how to deal with errors (noise) during the computation.

[In fact, the same conversion of hard-to-simulate system to useful one can be done for non-quantum systems. It is hard to simulate a system that contains N molecules, where N is the Avogadro number, even if the behavior of each molecule is simple. So we may get an efficient computing device if we arrange things in such a way that N different molecules perform (in parallel) N different computations.

This idea was proposed for DNA molecules ("biological computations"). However, in this case the possible speedup is linear (N times). Though N-times slowdown makes computation device completely useless, N-times speedup still is not enough to solve problems that require exponential search in reasonable time.]

Quantum computations without a quantum computer

There is another possibility to get something practically useful from quantum computations. Let us recall the universality phenomenon discussed above. One could expect that if we take a generic physical system that computes some function, either this function is easy to compute or this function is likely to be universal (in the sense that many other hard-to-compute functions are reducible to it.

So the plan is:

(a) to find some real system that "computes" something that is hard to predict;

(b) to find a reduction of practically important problem (like factoring) to the function computed by this system.

Then factoring can be done as follows: (1) given a number x that needs to be factored, we perform a (classical polynomial) computation and get some input X; (2) we feed X into our physical system and observe its behavior; (3) make conclusions about factors based on this behavior.

What kind of physical system do we need for this approach? Its desired properties are:

(a) Reproducibility: we should be able to construct similar systems in different labs from scratch and they should exhibit similar behavior (gave the same results for the same inputs).

(b) Scalability: we should be able to feed inputs that have thousands of bits (at least) in the system and the result should essentially depend on individual bits of the input.

(c) Clear theory: we should believe that the theory of the system is well understood and the only thing that prevents us from deriving its behavior from the first principles is computational complexity.

(d) Complexity: we should believe that computational complexity of predicting the behavior of the system is high.

May be some systems with these properties could be found in molecular biology (something like protein 3D-structure as function of DNA-sequence). Or some nano-electronic devices could exhibit this behavior.

Of course, all this is just science fiction, but if something like that could be done, it would be one more manifestation of the general universality principle: *if something is complex enough, it is*

universal.