Chapter 2

Computational Complexity

2.1 Computability

At the turn of the 20th century David Hilbert asked if it was possible to devise automatic theorem proving procedures that could decide whether a given mathematical statement is true or false. This led to great discoveries in mathematical logic, among them a formalization of the notion of computability of functions by computing devices. To understand the notion of computability of a function better, let's begin by considering some examples [**?**, p.10].

Let $f : \{0, 1, \ldots, 9\}^* \to \{0, 1, \ldots, 9\}^*$ be defined as follows:

$$f(x) = \begin{cases} 1 & x \text{ is an initial segment of the decimal expansion of } \pi, \\ 0 & \text{otherwise.} \end{cases}$$

For example, $f(1) = 0$, $f(314) = 1$, $f(32) = 0$, etc. This function is computable (think of a program to compute it on a PC).

Next, consider

$$g(x) = \begin{cases} 1 & x \text{ occurs somewhere in the decimal expansion of } \pi, \\ 0 & \text{otherwise.} \end{cases}$$

For example, $g(1) = 1$, $g(14) = 1$, but what about $g(863898)$? It is not clear how one might compute this function.

As a final example, consider

$$h(x) = \begin{cases} 1 & \text{there is run of at least } x \text{ consecutive 7's in the decimal expansion of } \pi, \\ 0 & \text{otherwise.} \end{cases}$$

This function is computable. Either there are arbitrarily long runs of 7's in which case $h(x) \equiv 1$; or, there is a finite maximum run-length m in which case $h(x) = 1$ for $0 \leq x \leq m$, and $h(x) = 0$ for $x > m$. In all cases, we know that *there exists an algorithm that computes* h; this is sufficient to call h computable. The fact that we are unable to pinpoint the exact algorithm does not make h uncomputable.

Following these examples, the following question arises naturally. Are all functions computable, or, are there functions that are demonstrably uncomputable? These questions cannot be answered without first fixing a computational model, in other words, specifying precisely what our computational capabilities are.

Which computational model should one use to define computability of functions? Clearly, the goal should be to use a model that is as strong as possible in the sense that it renders the largest class of functions computable. It is also important to realize that for the model to have any relevance it should be consistent with the laws of physics, so that in principle we can build such a computer.

A great deal of work has been done in devising computational models to this end. In a pioneering work, Alan Turing invented the Turing machine model that consists of an infinite tape (memory), a read-write head, and a finite-state machine (the program) that controls the motion of the head. The Turing machine is a primitive form of modern-day digital computer, except that it has infinite memory. Many failed attempts to design more powerful computational models than the Turing model have led to the

CHURCH-TURING THESIS: The intuitively conceived class of computable functions are precisely the class of functions that are computable by a Turing machine.

This statement cannot be proved because it refers to the undefined notion of 'intuitive computability.' The statement is refutable, however; if someone exhibited a more powerful model (that is consistent with the laws of physics) we would abondon this thesis.

If the Turing model is the strongest model that we can think of, can it compute all functions? The answer is 'no.' A Turing-uncomputable function is the HALTING function:

$$H(P,x) = \begin{cases} 1 & \text{if program } P \text{ eventually halts on input } x, \\ 0 & \text{if program } P \text{ runs forever on input } x. \end{cases}$$

A precise statement of the Halting problem and proof of its uncomputability can be found in Chapter 3 of Nielsen and Chuang. Many other Turing-uncomputable functions are known. In fact it is known that computable functions are a rarity, notwithstanding the fact that it is in general quite difficult to prove the uncomputability of a specific function.

2.1.1 Classical vs. Quantum Computability

The Turing model is classical in nature. It is designed with the classical laws of physics in mind. It does not utilize the capabilities offered by a quantum mechanical view of the world.

Could a computational model that utilizes quantum mechanical phenomena expand the domain the of computable functions? Dissapointingly, the answer is negative. Any quantum computer can be simulated on a classical Turing machine (albeit not very efficiently), so any function computable by a quantum computer is also computable classically.

The promise and appeal of quantum computing is of a more practical nature: to obtain a speed-up in solving certain problems not attainable by classical methods.

2.2 Computational Complexity Classes

The classification of functions as computable vs. uncomputable is of little practical significance. Almost always, we have a theoretically computable function but computing it in practice may not be feasible due to limitations on computational resources. There are three main types of computational resources: time, memory (space), and energy. Time and energy are non-recyclable, memory is recyclable. Computational complexity deals with determining the minimal computational resources necessary for the solution of a given problem. Problems are classified into complexity classes with respect to their resource requirements.

We saw that theoretical computability is insensitive to whether our computer has quantum mechanical parts in it or not. However, it is conceivable that quantum computers may offer savings in computational resources. So, in assessing the computational complexity of a specific problem, one must indicate clearly what kind of computer is under consideration: classical or quantum.

In this section we define various classical and quantum complexity classes. The key concept in such definitions is the notion of *problem size*. This is the length of a natural encoding of a problem instance measured in bits. For example if the problem is one of factoring an integer N, the problem size is $\log_2 N$ bits. If we wish to sort a database of N records in alphabetical order, the problem size is NK where K is the length of each record in bits.

Another concept in defining complexity classes is the notion of a *decision problem*. These are problems with a YES/NO answer. For example, given an integer N, is it prime? More general problems can usually be reduced to decision problems by binary search. Suppose, for example, that we wish to find the prime factors of an integer N. This problem can be solved through a sequence of decision problems of the type: *Does N have a prime factor less than k?*

2.2.1 *P*

Time is one of the primary resources that we keep track of in practical computational complexity. An algorithm for solving a problem is said to be a polynomial-time (polytime) algorithm if it solves every instance of the problem in time polynomially bounded in the size of the problem instance. We define P as the class of decision problems that can be solved by a polytime algorithm on a classical computer. Membership in P is regarded as the major criterion of practical solvability for a problem.

As example of a polytime problem, consider finding the minimum of N integers. The input size is of order N and a simple algorithm solves this problem in time of order N.

An important problem for which no (classical) polytime algorithm is known is FACTORING. We can find the prime factors of a given integer N by exhaustively trying to divide it by each integer smaller than N. The worst case running time of such an algorithm before finding a prime factor (or deciding that the number N is prime) is $\sqrt{N}$, which is exponential in the problem size. More efficient factoring algorithms are known but none is polytime. There is also no proof that no (classical) polytime algorithm exists for FACTORING.

2.2.2 *PSPACE*

This is the class of decision problems that can be solved on a classical computer using memory polynomially bounded in the size of the problem. No constraints are placed on the time it takes to solve the problem. It is obvious that $P \subseteq PSPACE$ since a polytime algorithm has no time to use super-polynomial space. It may be surprising to learn that we still do not know if *PSPACE* is strictly larger than P.

2.2.3 *NP*

Another very important class of problems is the class *NP*, where *N* stands for non-deterministic and P for polytime. Consider, e.g., a Boolean expression

$$f(x_1, x_2, x_3, x_4, x_5) = (x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_2 \vee \overline{x_3} \vee \overline{x_4}) \wedge (x_1 \vee x_3 \vee x_5).$$

Is there a truth assignment to the Boolean variables $x_1, \ldots, x_5$ such that $f = 1$? This is known as the SATISFIABILITY problem (SAT for short). There is no known polytime algorithm for solving SAT. On the other hand, it is possible to check in polytime whether a truth assignment satisfies $f = 1$. Thus, every YES instance of the SAT problem has a polytime *certificate* that consists of a proposed solution and a polytime algorithm for verifying the validity of the proposed solution. This also means that if we luckily guessed a solution to the problem, we could in polytime verify that the guess is correct.

A decision problem is said to be in *NP* if every YES instance of the problem has a polytime certificate. The problem SAT is in *NP*.

It is known that

$$P \subseteq NP \subseteq PSPACE.$$

It is not known if any of the inclusions is strict. It is widely conjectured that $P \neq NP$. It is possible (for all we know) that both inclusions are strict.

In 1971 Cook proved that SAT is at least as difficult as any other problem in *NP* in the following sense: Every problem in *NP* is polynomially-reducible to SAT. This means that if SAT is in *P* then $NP = P$. SAT is said to be *NP*-complete due to this property. Many problems that arise naturally in combinatorial optimization turn out to be *NP*-complete. The class of *NP*-complete problems is denoted by *NPC*.

It is conjectured that $NPC \neq P$. If this conjecture is true there must be some problems of intermediate difficulty between *P* and *NPC*. The class *NPI* is defined as the set of problems in *NP* that are neither in *NPC* nor in *P*. It is believed that *NPI* is non-empty and the FACTORING problem is a member of it.

2.2.4 *BPP*

BPP stands for *bounded-error probabilistic polytime*. This is the class of decision problems for which a classical computer equipped with a random-number generator finds the correct solution (YES or NO) in polytime with probability greater than 3/4. (The number 3/4 is arbitrary; any number strictly larger than 1/2 will do. The idea is to run the algorithm many times and pick the solution by majority vote.)

Probabilistic algorithms have proved very effective in some problems such as determining if a number is prime. (This may sound surprising since it is not known if FACTORING is in *BPP*.) We already saw an example that illustrates the effectiveness of probabilistic algorithms; namely, Deutsch's problem.

Clearly, $P \subseteq BPP$. The relation of *BPP* to *NP* is unknown.

2.2.5 *BQP*

BQP is defined as the class of decision problems that can be solved on a quantum computer with probability of success larger than 3/4 and running time bounded polynomially in problem size.

This is the analog of the classical class *BPP*. Since quantum computers are inherently probabilistic, there is no quantum analog of the class *P*. *BQP* is regarded as the class of problems that are effectively solvable on a quantum computer. FACTORING is an example of a problem in *BQP*.

The relation of *BQP* to *BPP* is unknown. So, strictly speaking, it is not yet a proven fact that quantum computers can enlarge the domain of effectively computable functions.

The relation of *BQP* to *NP* is also unknown. It is known however that $BQP \subseteq PSPACE$.

If one were to show that *BQP* is strictly larger than *BPP*, that would imply $P \neq PSPACE$, settling a long-standing open problem in classical complexity theory. This is an indication of the difficulty of the problem.

2.3 The Circuit Model of Computation

A computational model that is equivalent to the Turing model in computational power is the *circuit model*. There are classical and quantum circuit models. Classical models may be *irreversible* or *reversible*. Quantum models are necessarily reversible. We begin by considering irreversible classical circuits.

2.3.1 Classical Circuits

As we saw in earlier discussion, Boolean functions of the form

$$f : \{0,1\}^n \to \{0,1\}. \tag{2.1}$$

serve as primitives of computation. Solutions of arbitrary computational problems can be reduced to the computation of such functions. For example, a more general Boolean function

$$f : \{0,1\}^n \to \{0,1\}^m.$$

is made up of m 'coordinate' functions

$$f_i : \{0,1\}^n \to \{0,1\}$$

such that

$$f(x_1, \ldots, x_n) = (f_1(x_1, \ldots, x_n), \ldots, f_m(x_1, \ldots, x_n)).$$

Functions of type (**??**) can in turn be computed in terms of a few elementary functions, namely NOT, AND, and OR. These functions, called *gates*, are defined as follows. The NOT gate is a function of one variable

$$\mathrm{NOT}(x) = \overline{x} = 1 - x.$$

The AND gate is a function of two variables and is denoted by the operand $\wedge$.

$$\mathrm{AND}(x, y) = x \wedge y = x \cdot y.$$

The OR gate is also a function of two variables and is denoted by $\vee$.

$$\mathrm{OR}(x, y) = x \vee y = x + y - x \cdot y.$$

The definitions of AND and OR are extended to n bit inputs through the recursions

$$\mathrm{AND}(x_1, \ldots, x_n) = \mathrm{AND}(x_1, \ldots, x_{n-1}) \wedge x_n,$$

$$\mathrm{OR}(x_1, \ldots, x_n) = \mathrm{OR}(x_1, \ldots, x_{n-1}) \vee x_n.$$

We also write $x_1 \wedge \cdots \wedge x_n$ to denote $\mathrm{AND}(x_1, \ldots, x_n)$, and $x_1 \vee \cdots \vee x_n$ to denote $\mathrm{OR}(x_1, \ldots, x_n)$.

To see that arbitrary functions can be implemented in terms of these gates, consider the function f defined by the following truth table.

x_1	x_2	x_3	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

It can be verified easily that

$$f(x_1, x_2, x_3) = (\overline{x_1} \wedge \overline{x_2} \wedge x_3) \vee (x_1 \wedge \overline{x_2} \wedge x_2) \vee (x_1 \wedge x_2 \wedge x_3).$$

This formula is obtained by putting an AND clause for each combination of inputs that give $f = 1$ and connecting the AND clauses by the OR function. Clearly, any Boolean function can be expressed in this form, called the *disjunctive normal form* (DNF).

A set of gates is called *universal* if arbitrary Boolean functions can be synthesized in terms of them. We have shown above that the NOT, OR and AND gates are universal. There are other sets of universal gates. In fact, a single gate NAND is universal by itself. NAND (not AND) is defined as

$$\mathrm{NAND}(x, y) = \overline{x \wedge y} = \overline{x} \vee \overline{y}.$$

The universal set NOT, OR, and AND can be implemented out of NAND by

$$\mathrm{NOT}(x) = \mathrm{NAND}(x, x) = \mathrm{NAND}(x, 1),$$

$$\mathrm{OR}(x, y) = \mathrm{NAND}(\overline{x}, \overline{y}),$$

$$\mathrm{AND}(x, y) = \mathrm{NAND}(\mathrm{NAND}(x, y), 1).$$

Implicit in the above discussion of classical circuits is that we can generate copies of variables through an operation

$$\mathrm{COPY}(x) = xx$$

and that we can prepare constant bits 1 or 0 to be applied to gate inputs as we wish. (When we go to quantum circuits, we will see that the COPY operation cannot be carried out for arbitrary states because of the *no-cloning* theorem.)

To summarize, an arbitrary Boolean function can be computed using combinations of a few elementary gates. In actual physical implementation of a function $f(x_1, \ldots, x_n) = (y_1, \ldots, y_m)$ from n bits to m bits, a circuit is built consisting of the following elements:

- a number of gates (NOT, AND, OR, etc.),
- n input wires where inputs $x_1, \ldots, x_n$ are applied,
- m output wires where outputs $y_1, \ldots, y_m$ are read,
- ancillary bits 0 and 1 applied as necessary,
- interconnection of gates by a network of wires with permission for fan-out (COPY).

2.3.2 Circuit complexity

Circuits are memoryless. All signals in the circuit settle to steady-state values soon after the application of inputs. Thus, regardless of the depth of the circuit, the operation of a circuit has no time-complexity. Instead, we measure the complexity of a circuit by its size, i.e., the number of elementary gates in it.

This is in contrast to the Turing model, in which the hardware is fixed and the complexity is measured by the number of time steps. The circuit-complexity can be related to Turing time-complexity, however there is a subtlety here which we now discuss.

A family of circuits $\{C_n\}$ indexed by an integer n is said to be *uniform* if there exists a polytime(n) algorithm that runs on a Turing machine and generates a complete design of the circuit C_n. This notion captures the design complexity of circuits, which should not be overlooked in estimating the complexity of solving a problem using circuits.

It can be shown that a family of functions f_n is in class P with respect to the Turing model if and only if there exists a polytime(n) uniform circuit family C_n that computes f_n. Thus, the uniformity condition enables establishing a correspondence between complexity classes in the Turing and circuit models.

Bibliography

[1] M. Machtey and P. Young, *An Introduction to the General Theory of Algorithms.* Elsevier, NY: 1982.