Chapter 2
Algorithms and Computation

In this chapter, we first discuss the principles of algorithm and computation in general framework, common both in classical and quantum computers, then we go to the fundamental topics of a Turing machine, algorithm, computation, circuits, and NP-complete problems.

2.1 General Algorithms

An algorithm is a precise formulation of doing something. Algorithms play an important role in mathematics and in computers, and they are employed to accomplish specific tasks using data and instructions. The notion of an algorithm is old; there is, for example, the well known Euclid's algorithm for finding the greatest common divisor of two numbers. Let us exhibit Euclid's algorithm here.

Euclid's algorithm. Given two positive integers m and n, the task is to find their greatest common divisor, i.e., the largest positive integer which divides both m and n. Here m and n are interpreted as variables which can take specific values. Suppose that m is greater than n. The algorithm consists of three steps.

Step 1. Divide m by n and let r be the remainder.
Step 2. If $r = 0$, the algorithm halts; n is the answer.
Step 3. Replace the value of m by the current value of n, also replace the value of n by the current value of r and go back to Step 1.

An algorithm has *input*, i.e., some quantity which is given to it initially before the algorithm begins. In Euclid's algorithm, the input is a pair of two positive integers m and n. An algorithm also has *output*, i.e., some quantity which has a specified relation to the input. In Euclid's algorithm, the output is n in Step 2, which is the greatest common divisor of two given integers.

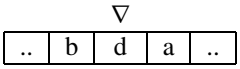
Exercise 2.1 Prove that the output of Euclid's algorithm is indeed the greatest common divisor.

M. Ohya, I. Volovich, *Mathematical Foundations of Quantum Information and Computation and Its Applications to Nano- and Bio-systems*,
Theoretical and Mathematical Physics,
DOI 10.1007/978-94-007-0171-7_2, © Springer Science+Business Media B.V. 2011

There are various approaches to precise formulation of the concept of an algorithm. There exist classical and quantum algorithms. One of modern precise formulations of the notion of a classical algorithm can be given by using Turing machines. Another approach to algorithms is based on the notion of circuits. Classical circuits and classical Turing machines are used as mathematical models of a classical computer. Quantum circuits and quantum Turing machines are mathematical models of a quantum computer.

The concept of the Turing machine was introduced by A.M. Turing in 1936 [757] for the study of limits of human ability to solve mathematical problems in a formal way. *Any classical algorithm can be implemented on a Turing machine* (this is the so-called A. Church's thesis).

A Turing machine has two main parts: a *tape* and a central unit with a *head* ∇ (see figure below).



The tape is infinite in both directions and is divided into squares. Each square of the tape holds exactly one of the symbols from a finite set of symbols (this finite set of symbols is called an alphabet). The central unit with the head is in one of the states from a finite set of states (the precise meaning of a state will be fixed later). The head sees at any moment of time one square of the tape and is able to read the content of the square as well as to write on the square. The input is written as a string (sequence) of symbols on the tape. The head starts in a prescribed state. In a single move, the Turing machine can read the symbol on the one square seen by its head, and based on that symbol and its current state, replace the symbol by a difference one, change its state, and move the head one square to the left, or one square to the right, or stay on the same square as before.

A sequence of moves is called a computation. For some pairs of states and symbols on the tape, the machine halts. In this case, symbols remaining on the tape form the output corresponding to the original input. A Turing machine accepts some input strings if it halts on it. The set of all accepted strings is called a language accepted by the Turing machine. Such languages are called recursively enumerable sets.

The Turing machine is a suitable model for the computational power of a classical computer. Its usefulness follows from the Church's thesis which may be reformulated as follows: *The computational power of the Turing machine represents a limit for any realizable classical computer.*

Let us indicate now one method which is general enough to include classical as well as quantum algorithms. Let us take two sets I and O. The set I will represent the input, and the set O represents the output of our computation. Suppose the sets I and O are parts of a larger set $\mathcal{X}$ which will represent configurations of computation. Let $G = \{g_1, \ldots, g_r\}$ be a finite set of functions g_i from $\mathcal{X}$ to $\mathcal{X}$. Such functions are called *gates* in computing and G is called the basis of gates. They form the primitive elements from which we will design an algorithm. For example, the gates can represent the basic logical operations such as AND, OR, and NOT as discussed in Chap. 1. Now let us be given a function f which maps the input set I to the

output set O. Our problem is to find a sequence of gates $A_G = \{g_{i_1}, g_{i_2}, \ldots, g_{i_k}\}$ which computes the function f in the sense that the function can be represented as a composition of gates, i.e., for any input $x \in I$ one has $f(x) = g_{i_1} \circ g_{i_2} \circ \cdots \circ g_{i_k}(x)$. The sequence A_G is called the algorithm or the program of computation using G.

Each input x in the set I defines a computational sequence, $x_0, x_1, \ldots$, as follows: $x_0 = x$, $x_1 = g_{i_1}(x_0), \ldots, x_m = g_{i_m}(x_{m-1}), \ldots$. One says that the computational sequence terminates in n steps if n is the smallest integer for which x_n is in O, and in this case it produces the output $y = x_n$ from x. One says that the algorithm computes the function $y = f(x)$.

A more general approach would be if one admits that the functions g_i and the function f are not defined everywhere (such functions are called partial functions) and that not every computational sequence terminates. Moreover, one can assume that the transition $x_m = g_{i_m}(x_{m-1})$ takes place with a certain probability (random walk) and that the output space O is a metric space with a metric τ. Then one says that the algorithm makes an approximate computation, in the order of ε, of a function $f(x)$ with a certain probability if one gets a bound $\tau(f(x), x_k) < \varepsilon$ for some $x_k \in O$.

To summarize, *the algorithm for the computation of the function f by using the prescribed set of gates is given by the following data $\{\mathcal{X}, I, O, G, A, f\}$ described above*.

In the considerations on this book, the set $\mathcal{X}$ for the classical Turing machine will be the set of all configurations of the Turing machine, and the gates g_i will form the transition function. For a classical circuit, the gates might, for example, be basic logical operations `AND`, `OR` and `NOT`. For a quantum circuit and for a quantum Turing machine, the set $\mathcal{X}$ might be the Hilbert space of quantum states and the gates g_i could be some unitary matrices and projection operators.

An important issue in computing is the computational complexity. One would like to minimize the amount of time and memory needed to produce the output from a given input. For an input x, let $t(x)$ be the number of steps until the computational sequence terminates. The computational time T of the algorithm is defined by

$$T(n) = \max_x \{t(x) : |x| = n\}$$

where $|x|$ is the length of the description of x. The actual length of the description depends on the model of computation. We are interested, of course, in minimizing the computational time $T(n)$.

2.2 Turing Machine

Now let us give a precise definition of the (classical) Turing machine. An *alphabet* A is a finite set of symbols (letters). For example, $A = \{0, 1\}$ or $A = \{\#, a_1, \ldots, a_m\}$. Let A, Q and Γ be three different alphabets. The alphabet $A = \{\#, a_1, \ldots, a_m\}$ is often called the tape alphabet. It should include the symbol # which is called the blank symbol.

Definition 2.2 A Turing machine M is a triple $M = \{Q, A, \delta\}$. Here $Q = \{q_0, q_1, \ldots, q_r, q_F\}$ is called the set of states. It should include the initial state q_0 and the final state q_F. δ is the transition function defined by $\delta : Q \times A \to Q \times A \times \Gamma$, where the set Γ consists of three symbols $\Gamma = \{L, S, R\}$.

Remark 2.3 In Γ, the letters L, S and R mean that the tape head goes to the left, stays, and moves right, respectively. It is useful to use the notation $\{-1, 0, +1\}$ instead of $\{L, S, R\}$, so that we may use both in the sequel when the later is considered useful.

The above function δ is not necessarily defined on all elements of Q. Such not everywhere defined functions are called partial functions. Normally, δ is defined on all elements of A and on all elements of Q except q_F.

The tape of the Turing machine is a sequence of squares, i.e., the one-dimensional lattice which can be enumerated by integers. Each square of the tape contains a symbol of A. The central unit of the machine is at each moment of time in one state $q \in Q$.

At the beginning, the input $x = (x_1, \ldots, x_n)$, $x_i \in A$, is contained in the squares labeled $0, 1, \ldots, n-1$, all other squares contain the blank symbol #. The head starts in the state q_0. If the head is in the state q and reads a on the tape, and if $\delta(q, a) = (q', a', \gamma)$ then the machine replaces the contents of the considered square by a', the new state of the head is q', and the head moves one step in direction γ. Here if $\gamma = R$ (or $+1$) then the head moves to the right, if $\gamma = L$ (or -1) then the head moves to the left, and if $\gamma = S$ (or 0) then the head stays on the same square. The computation stops if the head is in the final state q_F. The result of computation $y = (y_1, \ldots, y_s)$, where $y_i \in A$, can be read consecutively from the square starting at 0 until the first square containing the blank symbol #.

The Turing machine M transforms the input x into the output y. One can write $y = M(x)$. The computation does not necessarily halt on a given input x. Therefore, the function $M(x) = y$ is not necessarily defined on all inputs x. The following terminology is used. A *word* is a finitestring of symbols of the alphabet A, i.e., an ordered sequence $x_1 x_2 \cdots x_n$ where $x_i \in A$. The *length* $|w|$of the word $w = x_1 x_2 \cdots x_n$ is $|w| = n$. One can write $M(w) = v$ if the word w is an input, and the word v is the output for the Turing machine M. The function $v = M(w)$ is a partial function $M : A^* \to A^*$ from the set A^* of all words over the alphabet A to A^*.

Example 2.4 Let us design an adder, i.e., a Turing machine which makes addition of two natural numbers.

One takes $M = \{Q, A, \delta\}$ where the head alphabet $Q = \{q_0, q_1, q_F\}$ has three states, and the tape alphabet $A = \{\#, 1, +\}$ also has three symbols. If we want to compute the sum, say, of 2 and 3, then we will write the input word as $11 + 111$. The output should be 11111, i.e., 5. Our machine in this case acts as $M(11+111) = 11111$, and similarly for addition of arbitrary natural numbers n and m. The program

is given by the following transition function δ:

$$\begin{aligned} \delta(q_0, \#) &= (q_1, \#, L), \qquad & \delta(q_0, 1) &= (q_0, 1, R), \\ \delta(q_0, +) &= (q_0, 1, R), \qquad & \delta(q_1, 1) &= (q_F, \#, S). \end{aligned} \tag{2.1}$$

The program can be given also by the table below.

	#	1	+
q_0	$q_1, \#, L$	$q_0, 1, R$	$q_0, 1, R$
q_1		$q_F, \#, S$	

(2.2)

The machine starts in the state q_0 and looks at the left symbol 1. Then it moves to the right, finds the symbol + and prints instead of it the symbol 1. Then it goes up to the end, i.e., until reaches #, and finally moves one step to the left, prints # instead of 1 and halts.

Let us stress that the Turing machine is finite, it has only 9 symbols and the program is also finite, but the machine can make addition of two arbitrary natural numbers n and m.

Exercise 2.5 Design a Turing machine for the subtraction of two integers.

At every moment of time, the state of the Turing machine can be described as a *configuration* $C = (q, i, w)$ where q is the state of the head, the integer i is the number of the square on the tape to which the head looks, and w is the word on the tape formed by non-blank symbols. The program δ transforms one configuration into another. Computation on the Turing machine is a sequence of configurations.

The set of all words over A is denoted by A^*. A *language* L is a subset of A^*. A Turing machine *decides a language* L if it computes 1 for every input $A \in L$ and 0 if $A \notin L$. This means that the Turing machine computes the characteristic function of the language L. Here we assume that 0 and 1 belong to the alphabet A. A language is called *decidable* if there exists a Turing machine which computes its characteristic function. If a mathematical problem can be formulated as a decidable language then it is called a *solved problem in the sense of Turing*. Otherwise, the problem is called *unsolvable in the sense of Turing*. After the next section, we shall discuss an example of an unsolvable problem (the halting problem).

2.2.1 Gödel Encoding

Let us study the computational power of Turing machines. To this end, it is useful to enumerate all Turing machines in a special way. We have described the program of the Turing machine as a table or a matrix $(\delta(q_i, a_j))$ for all $q_i \in Q$ and $a_j \in A$. Now we will represent the program as a string. If $\delta(q, a) = (q', a', \gamma)$ then we will write the quintuplet (q, a, q', a', γ) where the first two symbols denote the arguments of

the function δ and the last three symbols denote its value. The program δ can be written as a string P of quintuplets.

For example, the program of the Turing machine from Example 2.4 can be written as

$$P = (q_0, \#, q_1, \#, L), (q_0, 1, q_0, 1, R), (q_0, +, q_0, 1, R), (q_1, 1, q_F, \#, S).$$

One can remove the brackets. Then we get

$$P = q_0, \#, q_1, \#, L, q_0, 1, q_0, 1, R, q_0, +, q_0, 1, R, q_1, 1, q_F, \#, S.$$

Now if one encodes the symbols of our Turing machine by natural numbers, for example, as

$$\begin{array}{lllll} q_0 \to 1, & q_1 \to 2, & q_F \to 3, & \# \to 4, & \\ 1 \to 5, & + \to 6, & R \to 7, & L \to 8, & S \to 9, \end{array}$$

then one can represent the program as a sequence of natural numbers:

$$P = 1, 4, 2, 4, 8, 1, \ldots, 9. \tag{2.3}$$

We would like to be able to represent or encode individual Turing machines as natural numbers. This would make it possible for one Turing machine to take another Turing machine (in encoded form) as its input. Of course, we assume a certain fixed encoding for the alphabets Q and A. Otherwise, the set of Turing machines will be uncountable because there is an uncountable number of encoded alphabets.

We will need some facts about prime numbers. *Prime numbers* are such natural numbers which are not divisible to other numbers except 1 and themselves.

Exercise 2.6 Prove that there are infinitely many prime numbers.

Every natural number $n \geq 2$ has a unique prime decomposition, $n = p_1^{i_1} p_2^{i_2} \cdots$, where $p_1 = 2$, $p_2 = 3, \ldots$. This decomposition leads to a one-to-one correspondence between finite sequences of natural numbers $(i_1, i_2, \ldots)$ and natural numbers n. In particular, one can encode the program as the following number:

$$2^1 3^4 5^2 7^4 11^8 \cdots = n.$$

Such an encoding is called the *Gödel encoding*, and n is called the *Gödel number* of the Turing machine M. It is evident that in this way we can enumerate all Turing machines $\{M_n\}$. The set of all Gödel numbers $\{n\}$ is a subset of natural numbers. Note that from this remark it follows that the set of all Turing machines is a countable set. Given a natural number, there is an algorithm for determining whether it is the Gödel number of a Turing machine.

Exercise 2.7 Prove that the number 49 is not a Gödel number.

2.2.2 The Halting Problem

Let us prove that there are uncomputable functions. Let $\{M_n\}$ be the set of all Turing machines where n is a Gödel number. Let us define the following function h: $h(n)$ is defined only for such Gödel numbers n for which M_n does not halt on input n, and for such n on has

$$h(n) = 0.$$

Theorem 2.8 *The function h is not Turing computable.*

Proof Let us assume that the function h is computable. Then there exists a Turing machine M_k with a Gödel number k that computes h. Let us consider what happens when M_k is presented with input k. The machine M_k either halts on the input k or it does not halt. Suppose first that M_k halts on input k, i.e., the machine computes the value of the function $h(k)$. But according to the definition of the function h, the value $h(k)$ is defined only if M_k does not halt on input k. We get the contradiction. Suppose now that M_k does not halt on input k. Then from the definition of h, one gets that $h(k)$ is defined, and moreover $h(k) = 0$. But since M_k does not halt on input k, this means that M_k does not compute the value $h(k)$. However, according to our assumption the function h has to be computed by M_k. Again we get the contradiction. Therefore, our assumption that the function h is computable cannot be true. □

2.2.3 Universal Turing Machine

A universal Turing machine is a machine which can simulate the behavior of any Turing machine whose programs are provided as input data to it. A universal Turing machine is comparable to a general purpose computer which can compile and execute any given input program. The universal Turing machine U will receive as input the description of a particular machine M in addition to a copy of an input x and will simulate the behavior of M when M starts on x. In other words, U receives (δ_M, x) as input, where δ_M is the string of quintuples of M and x is an arbitrary input for M. The universal Turing machine U then simulates the behavior of M when M starts on x. One can prove that there exists a universal Turing machine [197].

2.2.4 Partially Recursive Functions

Which functions are computable on Turing machines? This is the class of partially recursive functions. They are generated from basic functions by using generating rules. The basic functions are defined on the natural numbers. There are three basic functions:

(1) The successor function

$$s(x) = x + 1.$$

(2) The zero function

$$\mathbf{z}(x) = 0.$$

(3) The identity function

$$i(x) = x.$$

Similarly, the basic functions of several variables are defined by $n(x_1, \ldots, x_m) = 0$ and $i_j^m(x_1, \ldots, x_m) = x_j$.

Generating rules include composition, primitive recursion and minimization. For example, the function $f(x) = 1$ can be constructed from the basic functions by using the composition as follows: $f(x) = s(\mathbf{z}(x))$.

Let $g(x, y)$ be a given function of two variables and k be a natural number. We define a new function $r(x)$ by using g and the primitive recursion as:

$$\begin{aligned} r(0) &= k, \\ r(x+1) &= g\big(x, r(x)\big), \quad x \geq 0. \end{aligned}$$

Similarly for functions of several variables, let us take two functions $g(x_1, \ldots, x_{n+1})$ and $h(x_1, \ldots, x_{n-1})$. We define a new function $f(x_1, \ldots, x_n)$ by using g, h, and the primitive recursion as:

$$\begin{aligned} f(x_1, \ldots, x_{n-1}, 0) &= h(x_1, \ldots, x_{n-1}), \\ f(x_1, \ldots, x_{n-1}, y+1) &= g\big(x_1, \ldots, x_{n-1}, y, f(x_1, \ldots, x_{n-1}, y)\big), \quad y \geq 0. \end{aligned}$$

Example 2.9 The addition function $f(x, y) = x + y$ can be defined through the use of primitive recursion and composition as

$$\begin{aligned} f(0, y) &= i(y) = y, \\ f(x+1, y) &= s\big(f(x, y)\big). \end{aligned}$$

Exercise 2.10 Define the multiplication function $f(x, y) = xy$ from the basic functions by using composition and the primitive recursion.

The minimization produces a new function $f(x_1, \ldots, x_n)$ from a function $g(x_1, \ldots, x_{n+1})$ by the relation

$$f(x_1, \ldots, x_n) = \min\big\{y \mid g(x_1, \ldots, x_n, y) = 0\big\}.$$

Partially recursive functions are those functions which are generated from the basic functions by using a finite number of generating rules.

The following theorem describes the class of functions which are computable on Turing machines (for a proof, see, e.g., [757]).

Theorem 2.11 *The class functions computable on Turing machines is equivalent to the class of partially recursive functions.*

2.3 Boolean Functions, Gates and Circuits

2.3.1 Boolean Functions and Gates

In computations, it is often convenient to use the binary system and to reduce a problem to the study of Boolean functions. A Boolean function $f(x_1, \ldots, x_n)$ is a function of n variables where each variable takes values $x_i = 0$ or 1 and the function also takes values 0 or 1. If we denote $B = \{0, 1\}$ then the function f is a map $f : B^n (= \underbrace{B \times \cdots \times B}_{n}) \to B$.

There are three important Boolean functions which represent logical operations AND, OR and NOT. They are also called *gates* as before. The functions are:

(1) The conjunction

$$g_{\mathrm{AND}}(x_1, x_2) = x_1 x_2 = \min(x_1, x_2).$$

(2) The disjunction

$$g_{\mathrm{OR}}(x_1, x_2) = \max(x_1, x_2) = \begin{cases} 1, & \text{if } x_1 = 1 \text{ or } x_2 = 1, \\ 0, & \text{if } x_1 = 0 = x_2. \end{cases}$$

(3) The negation

$$g_{\mathrm{NOT}}(x) = \begin{cases} 1, & \text{if } x = 0, \\ 0, & \text{if } x = 1. \end{cases}$$

One also uses the notations $g_{\mathrm{AND}}(x_1, x_2) = x_1 \wedge x_2$, $g_{\mathrm{OR}}(x_1, x_2) = x_1 \vee x_2$ and $g_{\mathrm{NOT}}(x) = \overline{x}$. $\wedge$ and $\vee$ are often called *meet* and *join*, respectively. One has also the disjunction of several variables which is the composition of the binary disjunctions:

$$g_{\mathrm{OR}}(x_1, x_2, \ldots, x_n) = g_{\mathrm{OR}}\big(x_1, g_{\mathrm{OR}}\big(x_2, \ldots, g_{\mathrm{OR}}(x_{n-1}, x_n)\big)\big).$$

One has

$$g_{\mathrm{OR}}(x_1, x_2, \ldots, x_n) = \max(x_1, x_2, \ldots, x_n) = x_1 \vee x_2 \vee \cdots \vee x_n.$$

Similarly, for the conjunction of several variables one has

$$g_{\mathrm{AND}}(x_1, x_2, \ldots, x_n) = g_{\mathrm{AND}}\big(x_1, g_{\mathrm{AND}}\big(x_2, \ldots, g_{\mathrm{AND}}(x_{n-1}, x_n)\big)\big)$$

and

$$g_{\mathrm{AND}}(x_1, x_2, \ldots, x_n) = \min(x_1, x_2, \ldots, x_n) = x_1 x_2 \cdots x_n = x_1 \wedge x_2 \wedge \cdots \wedge x_n.$$

All logical circuits may be described in terms of the three fundamental elements.

If one denotes

$$x^\sigma = \begin{cases} \overline{x}, & \text{if } \sigma = 0, \\ x, & \text{if } \sigma = 1 \end{cases}$$

then it follows

$$x^\sigma = 1 \quad \Longleftrightarrow \quad x = \sigma. \tag{2.4}$$

Exercise 2.12 Prove the above relation (2.4).

Now let us prove that every Boolean function can be represented as a superposition of the fundamental elements.

Theorem 2.13 *Every Boolean function $f(x_1, \ldots, x_n)$ can be represented in the following form*

$$f(x_1, \ldots, x_n) = \bigvee_{\sigma_1, \ldots, \sigma_n} x_1^{\sigma_1} x_2^{\sigma_2} \cdots x_n^{\sigma_n} f(\sigma_1, \ldots, \sigma_n). \tag{2.5}$$

Proof For a given $(x_1, \ldots, x_n)$ one has either $f(x_1, \ldots, x_n) = 1$ or 0. First, let us assume $f(x_1, \ldots, x_n) = 1$. Then (2.4) leads to

$$x_1^{x_1} \cdots x_n^{x_n} f(x_1, \ldots, x_n) = 1,$$

and one gets (2.5).

Next let us assume $f(x_1, \ldots, x_n) = 0$. Note that $x_1^{\sigma_1} x_2^{\sigma_2} \cdots x_n^{\sigma_n} = 1$ only if $x_1 = \sigma_1, \ldots, x_n = \sigma_n$, which means that the right-hand side $x_1^{\bar{x}_1} x_2^{x_2} \cdots x_n^{x_n} f(x_1, \ldots, x_n) = 0$ form the assumption $f(x_1, \ldots, x_n) = 0$. □

The basic logical operations AND, OR and NOT are not independent. For example, the conjunction function g_{AND} can be represented as the superposition of the functions g_{OR} and g_{NOT} because one has

$$g_{\mathrm{AND}}(x, y) = g_{\mathrm{NOT}} \circ g_{\mathrm{OR}}\big(g_{\mathrm{NOT}}(x), g_{\mathrm{NOT}}(y)\big)$$

or

$$xy (\equiv x \wedge y) = \overline{\overline{x} \vee \overline{y}}. \tag{2.6}$$

Exercise 2.14 Prove the above relation (2.6).

Analogously the disjunction is represented by conjunction and negation:

$$x \vee y = \overline{\overline{x} \wedge \overline{y}}.$$

It follows that not only the set

$$\mathcal{A}_1 = \{g_{\mathrm{AND}}, g_{\mathrm{NOT}}, g_{\mathrm{OR}}\}$$

is a complete universal system (i.e., every Boolean function can be represented by this set $\mathcal{A}_2$) but also the set

$$\mathcal{A}_2 = \{g_{\mathtt{NOT}}, g_{\mathtt{AND}}\}$$

is a complete universal system for the Boolean system.

There are other complete universal systems. The following gates are important. They are the CNOT gate (*controlled* NOT *gate*) $g_{\mathtt{CNOT}} : B^2(= B \times B) \to B^2$,

$$g_{\mathtt{CNOT}}(x, y) = (x, x \oplus y)$$

and the *Toffoli gate* $g_{\mathtt{Toff}} : B^3(= B \times B \times B) \to B^3$:

$$g_{\mathtt{Toff}}(x, y, z) = (x, y, z \oplus xy).$$

Here $x \oplus y$ means addition in modulo 2:

$$0 \oplus 0 = 0, \qquad 0 \oplus 1 = 1 \oplus 0 = 1, \qquad 1 \oplus 1 = 0.$$

One can prove that the Toffoli gate is a universal gate for reversible computations.

2.3.2 Circuits

A representation is called a disjunctive normal form. It gives an example of a circuit. Every Boolean function can be written in such a form. However, this representation is not necessary the simplest representation of a function in terms of the fundamental ones. Some functions admit simpler representations.

Sometimes instead of the fundamental logical operations it is convenient to fix another set of functions as basic functions and represent any function in terms of these basic functions. A representation of a function using a given set of functions is called a *circuit*. We introduce some more notations before we give a formal definition of the circuit. For example, we shall write the function

$$f(x_1, x_2, x_3) = f_{\mathtt{AND}}\big(f_{\mathtt{OR}}(x_1, x_3), x_2\big)$$

as

$$f(x_1, x_2, x_3) = f^{(12)}_{\mathtt{AND}} \circ f^{(13)}_{\mathtt{OR}}(x_1, x_2, x_3)$$

where the superscript (13) in $f^{(13)}_{\mathtt{OR}}$ indicates that this function acts on the first and third arguments in (x_1, x_2, x_3) and the superscript (12) in $f^{(12)}_{\mathtt{AND}}$ indicates that the function acts on the first and second arguments.

In the more general case, if g is a function of k variables, $g : B^k \to B^m$ and $n \geq k$, then we define a function $g^{(i_1,\ldots,i_k)} : B^n \to B^{m+n-k}$ as

$$g^{(i_1,\ldots,i_k)}(x) = \big(g^{(i_1,\ldots,i_k)}(x_{i_1}, \ldots, x_{i_k}), x \setminus \{x_{i_1}, \ldots, x_{i_k}\}\big).$$

Here $x = (x_1, \ldots, x_n)$.

More precisely, if we introduce auxiliary variables $t_1 = x_1$ and $t_2 = f_{\mathrm{OR}}(x_2, x_3)$ then we can write $f(x_1, x_2, x_3) = f^{(12)}_{\mathrm{AND}}(t_1, t_2)$.

The functions $\{g_{\mathrm{AND}}, g_{\mathrm{OR}}, g_{\mathrm{NOT}}\}$ form a *universal basis* among Boolean functions in the sense that every Boolean function can be represented as a superposition of these functions. We reformulate Theorem 2.13 as

Theorem 2.15 *Every Boolean function $f(x_1, \ldots, x_n)$ can be written as a superposition of the three fundamental elements as*

$$f(x_1, \ldots, x_n) = g_{i_1}^{(\alpha_1)} \circ g_{i_2}^{(\alpha_2)} \circ \cdots \circ g_{i_k}^{(\alpha_k)}(x_1, \ldots, x_n).$$

Here g_i is one of the three fundamental elements, and the superscript (α) of $g_i^{(\alpha)}$ indicates on which arguments the element acts.

A circuit is a sequence of gates of the form where the functions g_i are some fixed Boolean functions.

Definition 2.16 A circuit C is the following set of data

$$C = \left\{G, f; f = g_{i_1}^{(\alpha_1)} \circ g_{i_2}^{(\alpha_2)} \circ \cdots \circ g_{i_k}^{(\alpha_k)}\right\},$$

where $G = \{g_1, \ldots, g_r\}$ is a finite set of Boolean functions (gates), f is a Boolean function of n variables, and one has the representation

$$f(x_1, \ldots, x_n) = g_{i_1}^{(\alpha_1)} \circ g_{i_2}^{(\alpha_2)} \circ \cdots \circ g_{i_k}^{(\alpha_k)}(x_1, \ldots, x_n).$$

2.4 Computational Complexity, P-problem and NP-problem

There are several problems which may not be solved effectively, namely, in polynomial time. The most fundamental problems are NP-problems and NP-complete problems. It is known that all NP-complete problems are equivalent, and an essential question is *whether there exists an algorithm to solve an NP-complete problem in polynomial time*. Such problems have been studied for decades now, and for each such problem all known algorithms have an exponential running time in the length of the input so far. A P-problem and an NP-problem are defined as follows:

Let n be the size of input.

Definition 2.17

(1) A P-problem is a problem with a time needed for solving it at worst a polynomial in n. Equivalently, it is a problem which can be recognized in a time polynomial in n by a deterministic Turing machine.

(2) An NP-problem is a problem that can be solved in polynomial time by a non-deterministic Turing machine (there must be a verifier that runs in polynomial time). This can be reexpressed as follows: Let us consider a problem of finding a solution of $f(x) = 0$. We can check in time polynomial in n whether x_0 is a solution of $f(x) = 0$, but we do not know whether we can find the solution of $f(x) = 0$ in time polynomial in n.
(3) An NP-complete problem is such an NP-problem to which any other NP-problem can be reduced in polynomial time.

Examples of a P-problem and an NP-complete problem

(1) *Euler closed path: P-problem.* Let $G = (V, E)$ be a graph, V be a set of vertices of the graph, and E be a set of edges of the graph. The problem of whether there exists a closed path from an arbitrary point v of V to v itself passing through all edges of E is called the *Euler closed path problem*. Let n be the number of all edges. Then it is known that the problem can be solved in polynomial time of order $O(n^3)$.
(2) *Hamilton closed path: NPC-problem.* In the above mentioned graph $G = (V, E)$, the problem of whether there exists a closed path from an arbitrary point v of V to v itself passing through all vertices of V is called the *Hamilton closed path problem*. It is not known whether an algorithm to solve this problem in polynomial time exists or not.
(3) *Traveling Salesman problem: NPC-problem.* Let $C = \{c_1, \ldots, c_n\}$ be the set of n cities, and let M be a given natural number. The distance $d(c_i, c_k)$ between two cities c_i, c_k is given, and the order of visiting all cities in $C = \{c_1, \ldots, c_n\}$ is denoted by π, namely, $c_{\pi(1)} \to \cdots \to c_{\pi(n)}$. Then the problem of whether there exists a π satisfying the following inequality

$$\sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)}) + d(c_{\pi(n)}, c_{\pi(1)}) \leq M$$

is called the *traveling salesman problem*.
(4) *SAT problem: NPC-problem.* Let $X \equiv \{x_1, \ldots, x_n\}$ be a set. Then x_k and its negation $\bar{x}_k$ $(k = 1, 2, \ldots, n)$ are called *literals*, and the set of all such literals is denoted by $X' \equiv \{x_1, \bar{x}_1, \ldots, x_n, \bar{x}_n\}$. The set of all subsets of X' is denoted by $\mathcal{F}(X')$, and an element $C \in \mathcal{F}(X')$ is called a *clause*. We take a truth assignment to all Boolean variables x_k. If we can assign the truth value to at least one element of C, then C is called *satisfiable*. When C is satisfiable, the truth value $t(C)$ of C is regarded as true, otherwise as false. Take the truth values as "true $\leftrightarrow$ 1", "false $\leftrightarrow$ 0". Then C is satisfiable iff $t(C) = 1$.

Let $B = \{0, 1\}$ be a Boolean lattice with the usual join $\vee$ and meet $\wedge$, and let $t(x)$ be the truth value of a literal x in X. Then the truth value of a clause C is written as $t(C) \equiv \bigvee_{x \in C} t(x)$.

Moreover, the set $\mathcal{C}$ of all clauses C_j $(j = 1, 2, \ldots, m)$ is called *satisfiable* iff the meet of all truth values of C_j is 1; $t(\mathcal{C}) \equiv \bigwedge_{j=1}^{m} t(C_j) = 1$. Thus the SAT problem is written as follows:

Definition 2.18 (SAT problem) Given a Boolean set $X \equiv \{x_1, \ldots, x_n\}$ and a set $\mathcal{C} = \{C_1, \ldots, C_m\}$ of clauses, determine whether $\mathcal{C}$ is satisfiable or not.

That is, this problem is whether there exists a truth assignment to make $\mathcal{C}$ satisfiable. It is known in usual algorithms that it takes polynomial time to check the satisfiability only when a specific truth assignment is given, but we cannot determine the satisfiability in polynomial time when an assignment is not specified.

"The SAT problem is an NP-complete" (Cook's theorem).

It is not known whether every problem in NP can be solved quickly—this is called the P = NP problem (a Millennium problem).

As special representations of the algorithms discussed in this chapter, we will discuss a quantum algorithm, generalized quantum algorithm and a chaotic quantum algorithm in Chaps. 11 and 14.

2.5 Notes

Conventional discussions on a Turing machine and an algorithm can be read in the book [188], in which there are several examples illustrating the moves of Turing machines in computation processes. For a more recent exposition, see [723]. The original paper discussing Gödel encoding, from the point of view of undecidability, is [293]. Fredkin–Toffoli gates used for the construction of computational circuits were introduced in [268]. A popular textbook introducing computational complexity and NP-problems is [278].

http://www.springer.com/978-94-007-0170-0

Mathematical Foundations of Quantum Information and Computation and Its Applications to Nano- and Bio-systems

Ohya, M.; Volovich, I.

2011, XX, 760 p., Hardcover
ISBN: 978-94-007-0170-0