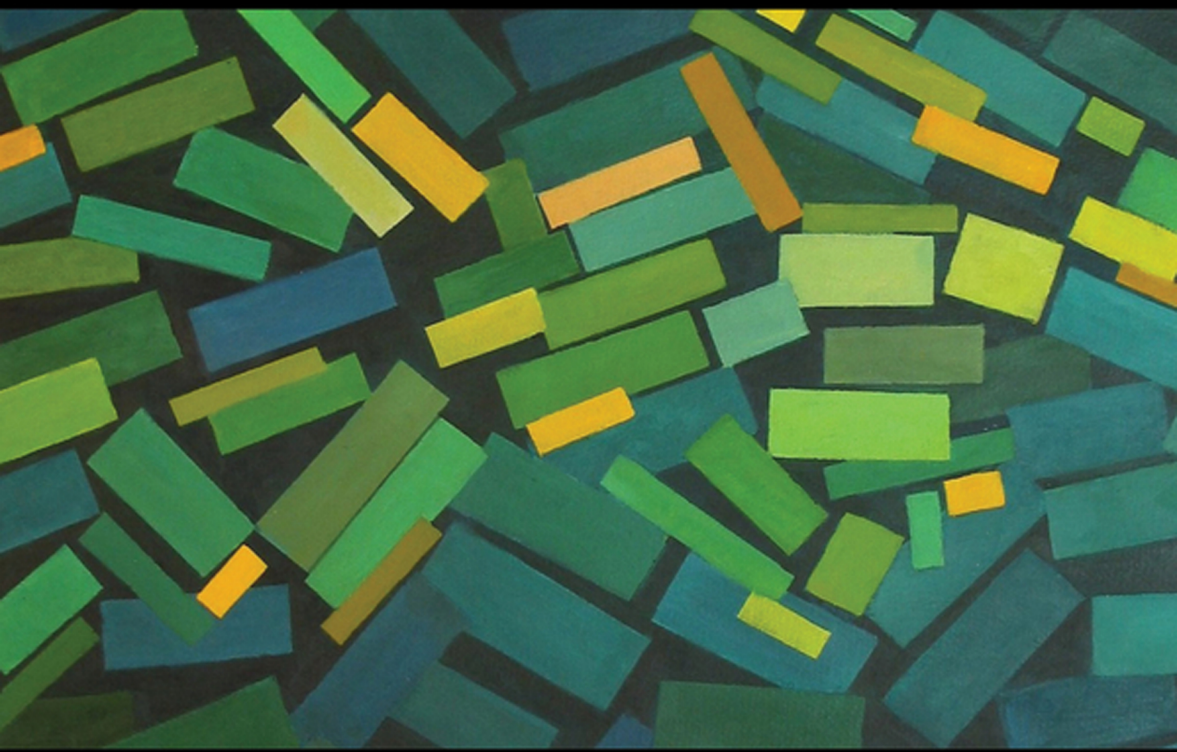
Wiley Series in Discrete Mathematics and Optimization



Ding-Zhu Du • Ker-I Ko

SECOND EDITION

Theory of Computational Complexity

WILEY

Theory of Computational Complexity

WILEY SERIES IN
DISCRETE MATHEMATICS AND OPTIMIZATION

A complete list of titles in this series appears at the end of this volume.

Theory of Computational Complexity

Second Edition

Ding-Zhu Du

Department of Computer Science
University of Texas at Dallas
Richardson, TX

Ker-I Ko

Department of Computer Science
National Chiao Tung University
Hsinchu, Taiwan

WILEY

Published by John Wiley & Sons, Inc., Hoboken, New Jersey
Published simultaneously in Canada

Library of Congress Cataloging-in-Publication Data:

Du, Dingzhu, author.
Theory of computational complexity / Ding-Zhu Du, Department of Computer Science, University of Texas at Dallas, Ann Arbor, Ml, Ker-l Ko, Department of Computer Science, State University of New York at Stony Brook, Stony Brook, NY. –Second edition.
pages cm
Includes bibliographical references and index.
ISBN 978-1-118-30608-6 (cloth)
1. Computational complexity. I. Ko, Ker-I, author. II. Title.
QA267.7.D8 2014
511.3′52–dc23
2013047839

Printed in the United States of America

ISBN: 9781118306086

10 9 8 7 6 5 4 3 2 1

Contents

Preface

Computational complexity theory has been a central area of theoretical computer science since its early development in the mid-1960s. The subsequent rapid development in the next three decades has not only established it as a rich exciting theory, but also shown strong influence on many other related areas in computer science, mathematics, and operations research. We may roughly divide its development into three periods. The works from the mid-1960s to the early 1970s paved a solid foundation for the theory of feasible computation. The Turing-machine-based complexity theory and the axiomatic complexity theory established computational complexity as an independent mathematics discipline. The identification of polynomial-time computability with feasible computability and the discovery of the *NP*-complete problems consolidated the *P* versus *NP* question as the central issue of the complexity theory.

From the early 1970s to the mid-1980s, research in computational complexity expanded at an exponential rate both in depth and in breadth. Various new computational models were proposed as alternatives to the traditional deterministic models. Finer complexity hierarchies and complexity classes were identified from these new models and more accurate classifications have been obtained for the complexity of practical algorithmic problems. Parallel computational models, such as alternating Turing machines and parallel random access machines, together with the *NC* hierarchy, provide a tool for the classification of the complexity of feasible problems. Probabilistic Turing machines are a model for the complexity theory of distribution-independent randomized algorithms. Interactive proof systems, an extension of probabilistic Turing machines, and communication complexity study the complexity aspect of distributed or interactive computing. The study of one-way functions led to a breakthrough in cryptography. A theory of average-case

completeness, based on the notion of distribution-faithful reductions, aims at establishing the foundation for the distribution-dependent average-case complexity. Boolean and threshold circuits are models for nonuniform complexity in which algebraic and topological methods have found interesting applications. Oracle Turing machines are the model for the theory of relativization in which combinatorial techniques meet recursion-theoretic techniques. Program-size complexity (or the Kolmogorov complexity) formalizes the notion of descriptive complexity and has strong connections with computational complexity. Although the central questions remain open, these developments demonstrate that computational complexity is a rich discipline with both a deep mathematical theory and a diverse area of applications.

Beginning in the mid-1980s, we have seen a number of deep surprising results using diverse sophisticated proof techniques. In addition, seemingly independent subareas have found interesting connections. The exponential lower bounds for monotone circuits and constant-depth circuits have been found using probabilistic and algebraic methods. The connection between constant-depth circuits and relativization led to the relativized separation of the polynomial-time hierarchy. The technique of nondeterministic iterative counting has been used to collapse the nondeterministic space hierarchy. The study of probabilistic reductions gave us the surprising result about the power of the counting class $\#P$ versus the polynomial-time hierarchy. Arithmetization of Boolean computation in interactive proof systems collapses the class of polynomial space to the class of sets with interactive proof systems. Further development of this research, together with techniques of coding theory, have led to strong negative results in combinatorial approximation.

As outlined above, complexity theory has grown fast both in breadth and in depth. With so many new computational models, new proof techniques, and applications in different areas, it is simply not possible to cover all important topics of the theory in a single book. The goal of this book is, therefore, not to provide a comprehensive treatment of complexity theory. Instead, we only select some of the fundamental areas that we believe represent the most important recent advances in complexity theory, in particular, on the P versus NP problem and present the complete treatment of these subjects. The presentation follows the approach of traditional mathematics textbooks. With a small number of exceptions, all theorems are presented with rigorous mathematical proofs.

We divide the subjects of this book into three parts, each representing a different approach to computational complexity. In Part I, we develop the theory of uniform computational complexity, which is based on the worst-case complexity measures on traditional models of Turing machines, both deterministic ones and nondeterministic ones. The central issue here is the P versus NP question, and we apply

the notions of reducibility and completeness to develop a complexity hierarchy. We first develop the notion of time and space complexity and complexity classes, in Chapter 1. Two basic proof techniques, simulation and diagonalization, including Immerman and Szelepcsényi's iterative counting technique, are presented. The knowledge of recursion theory is useful here but is not required. Chapter 2 presents the notion of *NP*-completeness, including Cook's theorem and a few well-known *NP*-complete problems. The relations between decision problems versus search problems are carefully discussed through the notion of polynomial-time Turing reducibility. Chapter 3 extends the theory of *NP*-completeness to the polynomial-time hierarchy and polynomial space. In addition to complete problems for these complexity classes, we also present the natural characterizations of these complexity classes by alternating Turing machines and alternating quantifiers. In Chapter 4, the structure of the class *NP* is analyzed in several different views. We present both the abstract proof that there exist problems in *NP* that are neither *NP*-complete nor in *P*, assuming *NP* does not collapse to *P*, and some natural problems as the candidates of such problems, as well as their applications in public-key cryptography. The controversial theory of relativization and their interpretations are also introduced and discussed in this chapter.

In Part II, we study the theory of nonuniform computational complexity, including the computational models of decision trees and Boolean circuits, and the notion of sparse sets. The nonuniform computational models grew out of our inability to solve the major open questions in the uniform complexity theory. It is hoped that the simpler structure of these nonuniform models will allow better lower bound results. Although the efforts so far are not strong enough to settle the major open questions in the area of uniform complexity theory, a number of nontrivial lower bound results have been obtained through new proof techniques. The emphasis of this part is therefore not on the subjects themselves but on the proof techniques. In Chapter 5, we present both the algebraic and the topological techniques to prove the lower bounds for decision trees of Boolean functions, particularly Boolean functions representing monotone graph properties. In Chapter 6, we present two exponential lower bound results on circuits using the approximation circuit technique and the probabilistic method. The notion of sparse sets links the study of nonuniform complexity with uniform complexity theory. This interesting interconnection between uniform and nonuniform complexity theory, such as the question of *NC* versus *P*, is also studied in Chapter 6. Then, we present, in Chapter 7, the works on the Hartmanis–Berman conjecture about the polynomial-time isomorphism of *NP*-complete problems, which provide further insight into the structure of the complexity class *NP*.

Part III is about the theory of probabilistic complexity, which studies the complexity issues related to randomized computation. Randomization in algorithms started in the late 1970s and has become increasingly popular. The computational model for randomized algorithms, the probabilistic Turing machine, and the corresponding probabilistic complexity classes are introduced in Chapter 8. The notion of probabilistic quantifiers is used to provide characterizations of these complexity classes, and their relations with deterministic and nondeterministic complexity classes are discussed. The counting problems and the complexity class $\#P$ may be viewed as an extension of probabilistic computation, and they are the main subjects of Chapter 9. Valiant's proof of $\#P$-completeness of the permanent problem, as well as Toda's theorem that all problems in the polynomial-time hierarchy are reducible to problems in $\#P$, are presented. The exponential lower bound of constant-depth circuits developed in Chapter 6 has an interesting application to the relativized separation of the complexity class $\#P$ from the polynomial-time hierarchy. This result is also presented in Chapter 9. Chapter 10 studies the notion of interactive proof systems, which is another extension of probabilistic computation. The collapse of the interactive proof systems hierarchy is presented, and the relations between the interactive proof systems and the Arthur–Merlin proof systems are discussed. Shamir's characterization of polynomial space by interactive proof systems is also presented as a prelude to the recent breakthrough on probabilistically checkable proofs. This celebrated result of Arora et al., that probabilistically checkable proofs with a constant number of queries characterize precisely the class NP, is presented in Chapter 11. We also present, in this chapter, the application of this result to various combinatorial approximation problems.

Although we have tried to include, within this framework, as many subjects in complexity theory as possible, many interesting topics inevitably have to be omitted. Two of the most important topics that we are not able to include here are program-size complexity (or the Kolmogorov complexity) and average-case completeness. Program-size complexity is a central theory that would provide a unified view of the other nonuniform models of Part II. However, this topic has grown into an independent discipline in recent years and has become too big to be included in this book. Interested readers are referred to the comprehensive book of Li and Vitányi (1997). Average-case completeness provides a different view toward the notion of distribution-independent average-case complexity and would complement the works studied in Part III about distribution-independent probabilistic complexity. This theory, however, seems to be still in the early development stage. Much research is needed before we can better understand its proof techniques and its relation to the worst-case complexity theory, and we reluctantly omit it here. We refer interested readers to Wang (1997) for a thorough review of this topic. Exercises at

the end of each chapter often include additional topics that are worth studying but are omitted in the main text owing to space limitations.

This book is grown out of authors' lecture notes developed in the past 10 years at the University of Minnesota and the State University of New York at Stony Brook. We have taught from these notes in several different ways. For a one-semester graduate course in which the students have had limited exposure to theory, we typically cover most of Part I plus a couple of chapters from either Part II or Part III. For better prepared students, a course emphasizing the recent advances can be taught based mainly on either Part II or Part III. Seminars based on some advanced materials in Parts II and III, plus recent journal papers, have also been conducted for Ph.D. students.

We are grateful to all our colleagues and students who have made precious suggestions, corrections, and criticism on the earlier drafts of this book. We are also grateful to the following institutions for their financial support in preparing this book: the National Science Foundation of the United States, National Science Council of Taiwan, National Natural Science Foundation of China, National 973 Fundamental Research Program of China, City University of Hong Kong, and National Chiao Tung University of Taiwan.

Ding-Zhu Du
Ker-I Ko

Notes on the Second Edition

The *PCP* characterization of the complexity class *NP* has been considered as one of the most important and insightful results in computational complexity theory. It was first proved in 1992, but the proof kept evolving in the past 20 years. Particularly, through the effort of Dinur and other researchers, we now have a combinatorial proof of this theorem that provides new different insight into this celebrated result. In the First Edition of this book, we presented the original proof. That proof was long and was based on algebraic coding theory, which made it hard to follow for students who are not familiar with this area. Due to the importance of the *PCP* theorem, in theory as well as in applications, we decided to replace the original proof by the new combinatorial proof that is based on the notion of expander graphs, a research area that has recently found many applications in computer science. We hope that the new proof will make this result more accessible to readers.

The work of the second author is partially supported by *Aiming for the Top University Program* of the National Chiao Tung University and Ministry of Education, Taiwan, R.O.C.

DING-ZHU DU
KER-I KO

Part I

Uniform Complexity

In P or not in P,
That is the question.
—William Shakespeare (?)

1

Models of Computation and Complexity Classes

O time! thou must untangle this, not I;
It is too hard a knot for me to untie.
— William Shakespeare

The greatest friend of truth is time.
— Charles Caleb Colton

The notions of algorithms and complexity are meaningful only when they are defined in terms of formal computational models. In this chapter, we introduce our basic computational models: deterministic Turing machines and nondeterministic Turing machines. Based on these models, we define the notion of time and space complexity and the fundamental complexity classes including P and NP. In the last two sections, we study two best known proof techniques, diagonalization and simulation, that are used to separate and collapse complexity classes, respectively.

1.1 Strings, Coding, and Boolean Functions

Our basic data structure is a string. All other data structures are to be encoded and represented by strings. A *string* is a finite sequence of

Theory of Computational Complexity, Second Edition. Ding-Zhu Du and Ker-I Ko.

symbols. For instance, the word *string* is a string over the symbols of English letters; the arithmetic expression "3 + 4 − 5" is a string over symbols 3, 4, 5, +, and −. Thus, to describe a string, we must specify the set of symbols to occur in that string. We call a finite set of symbols to be used to define strings an *alphabet*. Note that not every finite set can be an alphabet. A finite set S can be an alphabet if and only if the following condition holds.

Property 1.1 *Two finite sequences of elements in S are identical if and only if the elements in the two sequences are identical respectively in ordering.*

For example, $\{0, 1\}$ and $\{00, 01\}$ are alphabets, but $\{1, 11\}$ is not an alphabet because 11 can be formed by either 11 or (1 and 1).

Assume that Σ is an alphabet. A set of strings over the alphabet Σ is called a *language*. A collection of languages is called a *language class*, or simply a *class*.

The length of a string x is the number of symbols in the string x, denoted by $|x|$. For example, $|string| = 6$ and $|3 + 4 - 5| = 5$. For convenience, we allow a string to contain no symbol. Such a string is called the *empty string*, which is denoted by λ. So, $|\lambda| = 0$. (The notation $|\cdot|$ is also used on sets. If S is a finite set, we write $|S|$ to denote its cardinality.)

There is a fundamental operation on strings. The *concatenation* of two strings x and y is the string xy. The concatenation follows associative law, that is, $x(yz) = (xy)z$. Moreover, $\lambda x = x\lambda = x$. Thus, all strings over an alphabet form a monoid under concatenation.[1] We denote $x^0 = \lambda$ and $x^n = xx^{n-1}$ for $n \geq 1$.

The concatenation operation on strings can be extended to languages. The concatenation of two languages A and B is the language $AB = \{ab : a \in A, b \in B\}$. We also denote $A^0 = \{\lambda\}$ and $A^n = AA^{n-1}$ for $n \geq 1$. In addition, we define $A^* = \bigcup_{i=0}^{\infty} A^i$. The language A^* is called the *Kleene closure* of A. The Kleene closure of an alphabet is the set of all strings over the alphabet.

For convenience, we will often work only on strings over the alphabet $\{0, 1\}$. To show that this does not impose a serious restriction on the theory, we note that there exists a simple way of encoding strings over any finite alphabet into the strings over $\{0, 1\}$. Let X be a finite set. A one–one mapping f from X to Σ^* is called a *coding* (of X in Σ^*). If both X and $\{f(x) : x \in X\}$ are alphabets, then, by Property 1.1, f induces a coding from X^* to Σ^*. Suppose that X is an alphabet of n elements. Choose $k = \lceil \log n \rceil$ and choose a one–one mapping f from X to $\{0, 1\}^k$.[2] Note

[1] A set with an associative multiplication operation and an identity element is a *monoid*. A monoid is a *group* if every element in it has an inverse.

[2] Throughout this book, unless otherwise stated, log denotes the logarithm function with base 2.

that any subset of $\{0,1\}^k$ is an alphabet, and hence, f is a coding from X to $\{0,1\}^*$ and f induces a coding from X^* to $\{0,1\}^*$.

Given a linear ordering for an alphabet $\Sigma = \{a_1, \cdots, a_n\}$, the *lexicographic ordering* $<$ on Σ^* is defined as follows: $x = a_{i_1} a_{i_2} \cdots a_{i_m} < y = a_{j_1} a_{j_2} \cdots a_{j_k}$ if and only if either $[m < k]$ or $[m = k$ and for some $\ell < m$, $i_1 = j_1, \cdots, i_\ell = j_\ell$ and $i_{\ell+1} < j_{\ell+1}]$. The lexicographic ordering is a coding from natural numbers to all strings over an alphabet.

A coding from $\Sigma^* \times \Sigma^*$ to Σ^* is also called a *pairing function* on Σ^*. As an example, for $x, y \in \{0,1\}^*$ define $\langle x, y\rangle = 0^{|x|}1xy$ and $x \# y = x0y1x^R$, where x^R is the reverse of x. Then $\langle\cdot,\cdot\rangle$ and "#" are pairing functions on $\{0,1\}^*$. A pairing function induces a coding from $\underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n}$ to Σ^* by defining

$$\langle x_1, x_2, x_3, \ldots, x_n\rangle = \langle\cdots\langle\langle x_1, x_2\rangle, x_3\rangle, \ldots, x_n\rangle.$$

Pairing functions can also be defined on natural numbers. For instance, let $\iota : \{0,1\}^* \to \mathbb{N}$ be the lexicographic ordering function, that is, $\iota(x) = n$ if x is the nth string in $\{0,1\}^*$ under the lexicographic ordering (starting with 0). Then, we can define a pairing function on natural numbers from a pairing function on binary strings: $\langle n, m\rangle = \iota(\langle\iota^{-1}(n), \iota^{-1}(m)\rangle)$.

In the above, we have seen some specific simple codings. In general, if A is a finite set of strings over some alphabet, when can A be an alphabet? Clearly, A cannot contain the empty string λ because $\lambda x = x\lambda$. The following theorem gives another necessary condition.

Theorem 1.2 (McMillan's Theorem) *Let $s_1, \ldots, s_q$ be q nonempty strings over an alphabet of r symbols. If $\{s_1, \ldots, s_q\}$ is an alphabet, then*

$$\sum_{i=1}^{q} r^{-|s_i|} \le 1.$$

Proof. For any natural number n, consider

$$\Big(\sum_{i=1}^{q} r^{-|s_i|}\Big)^n = \sum_{k=n}^{n\ell} m_k r^{-k},$$

where $\ell = \max\{|s_1|, \ldots, |s_q|\}$ and m_k is the number of elements in the following set:

$$A_k = \{(i_1, \cdots, i_n) : 1 \le i_1 \le q, \ldots, 1 \le i_n \le q, k = |s_{i_1}| + \cdots + |s_{i_n}|\}.$$

As $\{s_1, \ldots, s_q\}$ is an alphabet, different vectors $(i_1, \ldots, i_n)$ correspond to different strings $s_{i_1} \ldots s_{i_n}$. The strings corresponding to vectors in A_k all have length k. Note that there are at most r^k strings of length k. Therefore, $m_k \le r^k$. It implies

$$\Big(\sum_{i=1}^{q} r^{-|s_i|}\Big)^n \le \sum_{k=n}^{n\ell} r^k r^{-k} = n\ell - (n-1) \le n\ell. \tag{1.1}$$

Now, suppose $\sum_{i=1}^{q} r^{-|s_i|} > 1$. Then for sufficiently large n, $(\sum_{i=1}^{q} r^{-|s_i|})^n > n\ell$, contradicting (1.1). ■

A *Boolean function* is a function whose variable values and function value are all in {TRUE, FALSE}. We often denote TRUE by 1 and FALSE by 0. In the following table, we show two Boolean functions of two variables, *conjunction* $\wedge$ and *disjunction* $\vee$, and a Boolean function of a variable, *negation* $\neg$.

x	y	$x \wedge y$	$x \vee y$	$\neg x$
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

All Boolean functions can be defined in terms of these three functions. For instance, the two-variable function *exclusive-or* $\oplus$ can be defined by

$$x \oplus y = ((\neg x) \wedge y) \vee (x \wedge (\neg y)).$$

For simplicity, we also write xy for $x \wedge y$, $x + y$ for $x \vee y$, and $\overline{x}$ for $\neg x$. A table like the above, in which the value of a Boolean function for each possible input is given explicitly, is called a *truth-table* for the Boolean function. For each Boolean function f over variables $x_1, x_2, \ldots, x_n$, a function $\tau : \{x_1, x_2, \ldots, x_n\} \to \{0, 1\}$ is called a *Boolean assignment* (or, simply, an *assignment*) for f. An assignment on n variables can be seen as a *binary* string of length n, that is, a string in $\{0,1\}^n$. A function $\tau : Y \to \{0, 1\}$, where $Y = \{x_{i_1}, x_{i_2}, \ldots, x_{i_k}\}$ is a subset of $X = \{x_1, x_2, \ldots, x_n\}$, is called a *partial assignment* on X. A partial assignment τ on n variables can be seen as a string of length n over $\{0, 1, *\}$, with $*$ denoting "unchanged." If $\tau : Y \to \{0, 1\}$ is a partial assignment for f, we write $f|_\tau$ or $f|_{x_{i_1} = \tau(x_{i_1}), \ldots, x_{i_k} = \tau(x_{i_k})}$ to denote the function obtained by substituting $\tau(x_{i_j})$ for x_{i_j}, $1 \le j \le k$, into f. This function $f|_\tau$ is a Boolean function on $X - Y$ and is called the *restriction* of f by τ. We say a partial assignment τ *satisfies* f or τ is a *truth assignment* for f, if $f|_\tau = 1$.[3]

The functions conjunction, disjunction, and exclusive-or all follow the commutative and associative laws. The distributive law holds for conjunction to disjunction, disjunction to conjunction, and conjunction to exclusive-or, that is, $(x + y)z = xz + yz$, $xy + z = (x + z)(y + z)$, and

[3] In the literature, the term *truth assignment* sometimes simply means a Boolean assignment.

$(x \oplus y)z = xz \oplus yz$. An interesting and important law about negation is de Morgan's law, that is, $\overline{xy} = \overline{x} + \overline{y}$ and $\overline{x + y} = \overline{xy}$. A *Boolean formula* is a formula over Boolean variables using operators $\vee$, $\wedge$, and $\neg$.

A *literal* is either a Boolean variable or the negation of a Boolean variable. An *elementary product* is a product of several literals. Consider an elementary product p and a Boolean function f. If $p = 1$ implies $f = 1$, then p is called an *implicant* of f. An implicant p is *prime* if no product of any proper subset of the literals defining p is an implicant of f. A prime implicant is also called a *minterm*. For example, function $f(x_1, x_2, x_3) = (x_1 + x_2)(\overline{x}_2 + x_3)$ has minterms $x_1\overline{x}_2$, x_1x_3, and x_2x_3. $x_1\overline{x}_2x_3$ is an implicant of f but not a minterm. The *size* of an implicant is the number of variables in the implicant. We let $D_1(f)$ denote the maximum size of minterms of f. A DNF (*disjunctive normal form*) is a sum of elementary products. Every Boolean function is equal to the sum of all its minterms. So, every Boolean function can be represented by a DNF with terms of size at most $D_1(f)$. For a constant function $f \equiv 0$ or $f \equiv 1$, we define $D_1(f) = 0$. For a nonconstant function f, we always have $D_1(f) \geq 1$.

Similarly, an *elementary sum* is a sum of several literals. Consider an elementary sum c and a Boolean function f. If $c = 0$ implies $f = 0$, then c is called a *clause* of f. A minimal clause is also called a *prime* clause. The size of a clause is the number of literals in it. We let $D_0(f)$ denote the maximum size of prime clauses of f. A CNF (*conjunctive normal form*) is a product of elementary sums. Every Boolean function is equal to the product of all its prime clauses, which is a CNF with clauses of size at most $D_0(f)$. For a constant function $f \equiv 0$ or $f \equiv 1$, we define $D_0(f) = 0$. For a nonconstant function f, we always have $D_0(f) \geq 1$.

The following states a relation between implicants and clauses.

Proposition 1.3 *Any implicant and any clause of a Boolean function f have at least one variable in common.*

Proof. Let p and c be an implicant and a clause of f, respectively. Suppose that p and c have no variable in common. Then we can assign values to all variables in p to make $p = 1$ and to all variables in c to make $c = 0$ simultaneously. However, $p = 1$ implies $f = 1$ and $c = 0$ implies $f = 0$, which is a contradiction. ■

1.2 Deterministic Turing Machines

Turing machines (TMs) are simple and yet powerful enough computational models. Almost all reasonable general-purpose computational models have been known to be equivalent to TMs, in the sense that they define the same class of computable functions. There are many variations of TMs studied in literature. We are going to introduce, in this section,

the simplest model of TMs, namely, the *deterministic Turing machine* (DTM). Another model, the *nondeterministic Turing machine* (NTM), is to be defined in the next section. Other generalized TM models, such as deterministic and nondeterministic oracle TMs, will be defined in later chapters. In addition, we will introduce in Part II other *nonuniform* computational models which are not equivalent to TMs.

A deterministic (one-tape) TM (DTM) consists of two basic units: the *control unit* and the *memory unit*. The control unit contains a finite number of states. The memory unit is a tape that extends infinitely to both ends. The tape is divided into an infinite number of tape squares (or, tape cells). Each tape square stores one of a finite number of tape symbols. The communication between the control unit and the tape is through a *read/write tape head* that scans a tape square at a time (See Figure 1.1).

A normal *move* of a TM consists of the following actions:

(1) Reading the tape symbol from the tape square currently scanned by the tape head;

(2) Writing a new tape symbol on the tape square currently scanned by the tape head;

(3) Moving the tape head to the right or left of the current square; and

(4) Changing to a new control state.

The exact actions of (2)–(4) depend on the current control state and the tape symbol read in (1). This relation between the current state and the current tape symbol and actions (2)–(4) is predefined by a *program*.

Formally, a TM M is defined by the following information:

(1) A finite set Q of states;

(2) An initial state $q_0 \in Q$;

(3) A subset $F \subseteq Q$ of accepting states;

(4) A finite set Σ of input symbols;

(5) A finite set $\Gamma \supset \Sigma$ of tape symbols, including a special blank symbol $\mathsf{B} \in \Gamma - \Sigma$; and

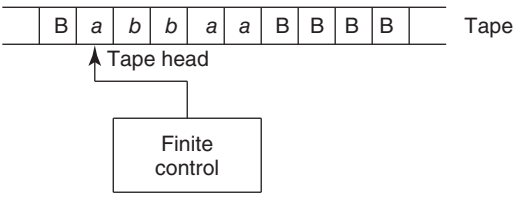


Figure 1.1 A Turing machine.

(6) A *partial* transition function δ that maps $(Q - F) \times \Gamma$ to $Q \times \Gamma \times \{\mathrm{L,R}\}$ (the *program*).

In the above, the transition function δ is a partial function, meaning that the function δ may be undefined on some pairs $(q, s) \in (Q - F) \times \Gamma$. The use of the initial state, accepting states, and the blank symbol is explained in the following text.

In order to discuss the notions of *accepting a language* and *computing a function* by a TM, we add some convention to the computation of a TM. First, we assume that initially an *input* string w is stored in the consecutive squares of the tape of M, and the other squares contain the blank symbol B. The tape head of M is initially scanning the leftmost square of the input w, and the machine starts at the initial state q_0. (Figure 1.1 shows the initial setting for a TM with input *abbaa*.) Starting from this initial configuration, the machine M operates move by move according to the transition function δ. The machine may either operate forever or halt when it enters a control state q and reads a tape symbol s for which $\delta(q, s)$ is undefined. If a TM M eventually halts at an accepting state $q \in F$ on input w, then we say M *accepts* w. If M halts at a nonaccepting state $q \notin F$ on input w, then we say M *rejects* w.

To formally define the notion of accepting an input string, we need to define the concept of *configurations*. A configuration α of a TM M is a record of all the information of the computation of M at a specific moment, which includes the current state, the current symbols in the tape, and the current position of the tape head. From this information, one can determine what the future computation is. Formally, a configuration of a TM M is an element (q, x, y) of $Q \times \Gamma^* \times \Gamma^*$ such that the leftmost symbol of x and the rightmost symbol of y are not B. A configuration (q, x, y) denotes that the current state is q, the current nonblank symbols in the tape are the string xy, and the tape head is scanning the leftmost symbol of y (when y is empty, the tape head is scanning the blank that is immediately to the right of the rightmost symbol of x).[4] Assuming $Q \cap \Gamma = \emptyset$, we also write xqy to stand for (q, x, y).

We now generalize the transition function δ of a TM M to the *next configuration function* $\vdash_M$ (or, simply $\vdash$ if M is understood) defined on configurations of M. Intuitively, the function $\vdash$ maps each configuration to the next configuration after one move of M. To handle the special nonblank requirement in the definition of configurations, we define two simple functions: $\ell(x)$ = the string x with the leading blanks removed and $r(x)$ = the string x with the trailing blanks removed. Assume that (q_1, x_1, y_1) is a configuration of M. If y_1 is not the empty string, then let $y_1 = s_1 y_2$ for some $s_1 \in \Gamma$ and $y_2 \in \Gamma^*$; if $y_1 = \lambda$, then let $s_1 = \mathrm{B}$ and

[4] The nonblank requirement for the leftmost symbol of x and for the rightmost symbol of y is added so that each configuration has a unique finite representation.

$y_2 = \lambda$. Then, we can formally define the function $\vdash$ as follows (we write $\alpha \vdash \beta$ for $\vdash(\alpha) = \beta$):

Case 1. $\delta(q_1, s_1) = (q_2, s_2, \mathrm{L})$ for some $q_2 \in Q$ and $s_2 \in \Gamma$. If $x_1 = \lambda$, then let $s_3 = \mathsf{B}$ and $x_2 = \lambda$; otherwise, let $x_1 = x_2 s_3$ for some $x_2 \in \Gamma^*$ and $s_3 \in \Gamma$. Then, $(q_1, x_1, y_1) \vdash (q_2, \ell(x_2), r(s_3 s_2 y_2))$.

Case 2. $\delta(q_1, s_1) = (q_2, s_2, \mathrm{R})$ for some $q_2 \in Q$ and $s_2 \in \Gamma$. Then, $(q_1, x_1, y_1) \vdash (q_2, \ell(x_1 s_2), r(y_2))$.

Case 3. $\delta(q_1, s_1)$ is undefined. Then, $\vdash$ is undefined on (q_1, x_1, y_1).

Now we define the notion of the computation of a TM. A TM M *halts* on an input string w if there exists a finite sequence of configurations $\alpha_0, \alpha_1, \ldots, \alpha_n$ such that

(1) $\alpha_0 = (q_0, \lambda, w)$ (this is called the *initial configuration* for input w);

(2) $\alpha_i \vdash \alpha_{i+1}$ for all $i = 0, 1, \ldots, n-1$; and

(3) $\vdash(\alpha_n)$ is undefined.

A TM M *accepts* an input string w if M halts on w and, in addition, the halting state is in F, that is, in (3) above, $\alpha_n = (q, x, y)$ for some $q \in F$ and $x, y \in \Gamma^*$. A TM M *outputs* $y \in \Sigma^*$ on input w if M halts on w and, in addition, the final configuration α_n is of the form $\alpha_n = (q, \lambda, y)$ for some $q \in F$.

Example 1.4 We describe a TM M that accepts the strings in $L = \{a^i b a^j : 0 \le i \le j\}$. The machine M has states $Q = \{q_0, q_1, \ldots, q_5\}$, with the initial state q_0 and accepting state q_5 (i.e., $F = \{q_5\}$). It accepts input symbols from $\Sigma = \{a, b\}$ and uses tape symbols in $\Gamma = \{a, b, c, \mathsf{B}\}$. Figure 1.2 is the transition function δ of M.

It is not hard to check that M halts at state q_5 on all strings in L, that it halts at a state q_i, $0 \le i \le 4$, on strings having zero or more than one b, and that it does not halt on strings $a^i b a^j$ with $i > j \ge 0$. In the following, we show the computation paths of machine M on some inputs (we write $x q_i y$ to denote the configuration (q_i, x, y)):

δ	a	b	c	B
q_0	q_1, c, R	$q_4, \mathsf{B}, \mathrm{R}$	$q_0, \mathsf{B}, \mathrm{R}$	
q_1	q_1, a, R	q_2, b, R		
q_2	q_3, c, L		q_2, c, R	$q_2, \mathsf{B}, \mathrm{R}$
q_3	q_3, a, L	q_3, b, L	q_3, c, L	$q_0, \mathsf{B}, \mathrm{R}$
q_4	q_4, a, R		$q_4, \mathsf{B}, \mathrm{R}$	$q_5, \mathsf{B}, \mathrm{R}$

Figure 1.2 The transition function of machine M.

On input *abaa*: $q_0abaa \vdash cq_1baa \vdash cbq_2aa \vdash cq_3bca \vdash q_3cbca \vdash q_3\mathsf{B}cbca \vdash q_0cbca \vdash q_0bca \vdash q_4ca \vdash q_4a \vdash aq_4 \vdash a\mathsf{B}q_5$.

On input *aaba*: $q_0aaba \vdash cq_1aba \vdash caq_1ba \vdash cabq_2a \vdash caq_3bc \vdash cq_3abc \vdash q_3cabc \vdash q_3\mathsf{B}cabc \vdash q_0cabc \vdash q_0abc \vdash cq_1bc \vdash cbq_2c \vdash cbcq_2 \vdash cbc\mathsf{B}q_2 \vdash cbc\mathsf{BB}q_2 \vdash \cdots$.

On input *abab*: $q_0abab \vdash cq_1bab \vdash cbq_2ab \vdash cq_3bcb \vdash q_3cbcb \vdash q_3\mathsf{B}cbcb \vdash q_0cbcb \vdash q_0bcb \vdash q_4cb \vdash q_4b$. □

The notion of computable languages and computable functions can now be formally defined. In the following, we say f is a *partial function* defined on Σ^* if the domain of f is a subset of Σ^*, and f is a *total function* defined on Σ^* if the domain of f is Σ^*.

Definition 1.5 *(a) A language A over a finite alphabet Σ is* recursively enumerable (r.e.) *if there exists a TM M that halts on all strings w in A and does not halt on any string w in $\Sigma^* - A$.*

(b) A language A over a finite alphabet Σ is computable *(or,* recursive*) if there exists a TM M that halts on all strings w in Σ^*, accepts all strings w in A and does not accept any string w in $\Sigma^* - A$.*

(c) A partial function f defined from Σ^* to Σ^* is partial computable *(or,* partial recursive*) if there exists a TM M that outputs $f(w)$ on all w in the domain of f and does not halt on any w not in the domain of f.*

(d) A (total) function $f : \Sigma^* \to \Sigma^*$ is computable *(or,* recursive*) if it is partial computable (i.e., the TM M that computes it halts on all $w \in \Sigma^*$).*

For each TM M with the input alphabet Σ, we let $L(M)$ denote the set of all strings $w \in \Sigma^*$ that are accepted by M. Thus, a language A is recursively enumerable if and only if $A = L(M)$ for some TM M. Also, a language A is recursive if and only if $A = L(M)$ for some TM M that halts on all inputs w.

Recursive sets, recursively enumerable sets, partial recursive functions, and recursive functions are the main objects studied in *recursive function theory*, or, *recursion theory*. See, for instance, Rogers (1967) for a complete treatment.

The above classes of recursive sets and recursively enumerable sets are defined based on the model of deterministic, one-tape TMs. As TMs look very primitive, the question arises whether TMs are as powerful as other machine models. In other words, do the classes of recursive sets and recursively enumerable sets remain the same if they are defined based on different computational models? The answer is yes, according to the famous Church–Turing Thesis.

Church–Turing Thesis. *A function computable in any reasonable computational model is computable by a TM.*

What is a *reasonable computational model*? Intuitively, it is a model in which the following conditions hold:

(1) The computation of a function is given by a set of finite instructions.

(2) Each instruction can be carried out in this model in a finite number of steps or in a finite amount of time.

(3) Each instruction can be carried out in this model in a deterministic manner.[5]

As the notion of *reasonable computational models* in the Church–Turing Thesis is not well defined mathematically, we cannot prove the Church–Turing Thesis as a mathematical statement but can only collect mathematical proofs as evidence to support it. So far, many different computational models have been proposed and compared with the TM model, and all *reasonable* ones are proven to be equivalent to TMs. The Church–Turing Thesis thus remains trustworthy.

In the following, we show that multi-tape TMs compute the same class of functions as one-tape TMs. A *multi-tape TM* is similar to a one-tape TM with the following exceptions. First, it has a finite number of tapes that extends infinitely to the both ends. Each tape is equipped with its own tape head. All tape heads are controlled by a common finite control. There are two special tapes: an *input tape* and an *output tape*. The input tape is used to hold the input strings only; it is a read-only tape that prohibits erasing and writing. The output tape is used to hold the output string when the computation of a function is concerned; it is a write-only tape. The other tapes are called the *storage tapes* or the *work tapes*. All work tapes are allowed to read, erase, and write (see Figure 1.3).

Next, we allow each tape head in a multi-tape TM, during a move, to stay at the same square without moving to the right or the left. Thus, each move of a k-tape TM is defined by a partial transition function δ that maps $(Q - F) \times \Gamma^k$ to $Q \times \Gamma^k \times \{\mathrm{L}, \mathrm{R}, \mathrm{S}\}^k$ (where S stands for *stay*). The initial setting of the input tape of the multi-tape TM is the same as that of the one-tape TM, and other tapes of the multi-tape TM initially contain only blanks. The formal definition of the computation of a multi-tape TM on an input string x and the concepts of accepting a language and computing a function by a multi-tape TM can be defined similar to that of a one-tape TM. We leave it as an exercise.

[5] By condition (1), we exclude the nonuniform models; by condition (2), we exclude the models with infinite amount of resources; and by condition (3), we exclude the nondeterministic models and probabilistic models. Although they are considered unreasonable, meaning probably not realizable by reliable physical devices, these nonstandard models remain as interesting mathematical models and will be studied extensively in the rest of the book. In fact, we will see that the Church–Turing Thesis still holds even if we allow nondeterministic or probabilistic instructions in the computational model.