COMPUTATIONAL COMPLEXITY

Osamu Watanabe
Tokyo Institute of Technology, Tokyo, Japan

Keywords: {deterministic, randomized, nondeterministic, circuit}computation model, {time, space, circuit size, circuit depth}complexity measure, complexity class, separation by diagonalization, inclusion by simulation, reducibility, completeness, combinatorial lower bound, monotone circuit complexity, one-way function, pseudo random bit generator, derandomization

Contents

Summary

In this chapter, we survey basic notions and fundamental theorems and questions in computational complexity theory. We will also explain some of the topics that have become active more recently. Note that there are many interesting research topics and important results in computational complexity theory. For these and the technical details omitted in this chapter, it is recommended to refer the textbooks given in the bibliography.

1. Introduction

Computational complexity theory is the field of theoretical computer science that studies the complexity of problems for answering questions like “How fast can we solve this problem?” In general, “resource bounded computability” is the main subject of computational complexity theory. Note that “computability” itself had been a topic of interest in mathematics long before the emergence of modern computers; indeed, the mathematical formulation of the “computability” notion lead to the invention of programmable digital computers. On the other hand, the efficiency of computation had not been considered seriously before the invention of the computer, but its importance was recognized as soon as computers were being used. Thus one could say that computational complexity theory is a field that has started with computers. In this chapter, we will survey basic notions, fundamental theorems and questions, and some

recent results in computational complexity theory. Before starting technical explanation, let us first agree on a framework that is assumed common to computational complexity theory.

In computational complexity theory, we consider only problems that are completely and formally specified. But even among mathematically specified problems, a problem like "Solve Riemann's conjecture" is not of our interest. Instead, we consider a problem asking for an answer to a given *instance* of the problem, i.e., a problem of computing some given function on a given input. For example, a problem is specified in the following way:

Max. Clique Problem (abbrev. CLIQUE)
Instance: An undirected graph G.
Question: What is the size of the largest clique (i.e., complete subgraph) of G?

Also for our technical convenience, we assume that input instances and outputs are encoded as binary strings, i.e., finite strings over the alphabet $\{0,1\}$. (We use $\{0,1\}^n$ and $\{0,1\}^*$ to denote respectively the set of binary strings of length n and the set of finite binary strings.) In the above problem, for example, we assume that the input graph G with n vertices is given as its adjacency matrix, which is expressed as n^2 bits. Of curse, there are various ways to encode a given object. Here we assume some "reasonably efficient" encoding. For example, for encoding an integer n, we usually use the standard binary code of n. But we may also encode n as a decimal notation and then encode each digit by using, e.g., the ASCII code. The difference is only some constant factor, and we usually ignore any constant (or small) difference. On the other hand, we may also encode n by a string 0^n, i.e., a sequence of 0's of length n. But this code is exponentially longer than the binary code, and so, it is not used unless required for some specific purpose.

Given a problem, one would like to ask the amount of "resource" (most typically "time") necessary for solving the problem when the most efficient algorithm is used, and this is the basic question on computational complexity. Of course, algorithm's running time heavily depends on the computer being used, and clearly, we cannot consider all existing computers. Thus, for discussing computational complexity, we use abstract machine models that represent (some aspects of) current computers. Note also that there are various kinds of resources, though "time" is usually the most important one. (We will see some typical machine models and their resources in the next section.)

Another important convention assumed in computational complexity theory is that complexity, i.e., the efficiency of algorithms or the difficulty of problems, is measured in terms of the "size" of input instances. This is because we are usually interested in how the complexity grows when instance size gets increased. There are various ways to define "instances size", and one should choose an appropriate one depending on what one wants to analyze. On the other hand, for general discussion, we may safely use the length, i.e., the number of bits, of an input string as instance size. (Recall that we assumed that each input instance is encoded as a binary string.) Throughout this chapter, "instance size" will mean to the length of the input string.

Finally, we propose throughout this chapter to consider only "decision problems", where a *decision problem* is to decide whether a given instance is 'yes' or 'no'. That is, we consider only problems of computing some function whose output is either 1 (meaning 'yes') or 0 (meaning 'no'). This simplifies our discussion because we can use simpler machine models and we do not have to worry about output length. What is more important is that this restriction is not essential because any problem can be converted to same decision problem with almost the same complexity. For example, for the CLIQUE problem mentioned above we can consider the following decision version.

Max. Clique Problem (CLIQUE)
Instance: An undirected graph G and an integer k.
Question: Does G have a clique of size $\geq k$?

These two versions of CLIQUE have almost the same complexity. Certainly, we can easily convert any algorithm for the previous CLIQUE to the one for the decision version. By using the standard binary search strategy, we can easily solve the previous CLIQUE by using any algorithm for its decision version. This type of relationship always holds in general. Therefore, unless complexity analysis is affected by some subtle difference between non-decision and decision version, decision problems are usually considered in computational complexity theory, and we will follow this convention here.

Notice that any decision problem is completely specified by the set of 'yes' instances of the problem, or more precisely, the set of binary strings encoding those 'yes' instances. Thus, we usually identify a problem with its 'yes' instances. For example, CLIQUE is the set of pairs (G,k) such that G has a clique of size $\geq k$.

In this chapter, journal publication year is used for stating the year of each result so long as it is published in some journal.

2. Machine Models and Complexity Measures

We first define our method of discussing the efficiency of a given algorithm, i.e., the amount of "resource" that the algorithm consumes. For this, we need to specify a machine on which the algorithm is executed, a type of resource that we want to measure, and the method of discussing the efficiency in total. In the following, we will explain these points.

First consider machineries for computation, often called *machine models or computation models.* Many machine models have been proposed, but we explain here four typical models: deterministic Turing machine, randomized Turing machines, nondeterministic Turing machines, and families of Boolean circuits. Since resources are defined depending on machine models, these are defined along with each model.

Deterministic Turing Machines (*Det. TMs*)

Turing machine (TM in short) is the most commonly used machine model that has been used by researchers in theoretical computer science. In particular, *multi-tape*

Deterministic Turing machine (det. TM in short) has been used as a model for normal sequential computation devices.

Let us recall briefly the definition of det. TM. A multi-tape Deterministic TM consists of one work-tape, one read-only input-tape, one write-only output-tape, and an internal state taking an element in some finite set. (For a model of algorithms solving decision problems, we do not need to use the output-tape; we can simply use two special halting states, i.e., the accepting (yes) and rejecting (no) states, for indicating machine's output.) A TM determines the next *move* from the current state and the symbols of the cells currently scanned by the machine's tape heads. In one move, the machine writes a symbol on the currently scanned cell on the work-tape, moves the work-tape and head to the right or left, and changes the internal state. It may also write a symbol on the output-tape and move the output-tape head to the right. For a given input x, the machine starts its computation with x on its input tape and blank symbols on the other part of the tapes. The computation terminates at the time when internal state becomes one of the halting states. The output of M on input x, which we denote $M(x)$, is the string written on the output-tape at termination.

By using TM, we can naturally define the notice of "computation time" and "memory space", the most important resources in computational complexity theory. For a TM M and for a given input x, the (*computation* or *running*) *time* of M on x, which we denote $\text{time}_M(x)$, is the number of moves that the machine takes form the start of the computation with this input, to its termination. The (*memory*) *space*, denoted by $\text{space}_M(x)$, is the number of tape cells of the *working tape* visited during the computation. This is the formal definition of "time" and "space". But in the following, for any deterministic algorithm A, we simply write as $\text{time}_A(x)$ and $\text{space}_A(x)$ referring to the time and space used by A on input x. By these we formally mean the time and space of some det. TM that "appropriately" implements the algorithm A. Note that, however, one does not have to always implement a given algorithm as a TM for discussing, e.g., its computation time. Usually it is enough to describe the algorithm with one's favorite programming language such as Pascal, C, etc, or with even less formal description. This is because it is tedious but conceptually easy to show that each instruction used in the description can be implemented as a sequence of a finite (or a small) number of TM's moves. That is, the difference is within some constant factor and usually such difference can be ignored in the analysis. (It may be the case that running time differs by $O(\log n)$ factor between TMs and high level machineries.)

Randomized Turing Machines (*Randomized TMs*)

We can sometimes design an efficient algorithm if we are allowed to use "randomness" in the computation. In fact, researchers have proposed many efficient randomized algorithms, algorithms that make use of a random bit sequence, i.e., an outcome of coin flipping. *Randomized Turing machine* (randomized TM in short) is a formal model for such algorithms.

A randomized TM is the same as a deterministic one except that it has a special state called a coin-flipping state. For the coin-flipping state (more precisely, for the

configuration having the coin-flipping state as the current internal state), we can define two different moves. Then in the execution, one of these two moves is chosen as an actual move *randomly* with the same probability. Due to this randomness, randomized TM M may execute differently on the same input x. We call each possible execution a *computation path* of M on input x. Notice that a computation path is uniquely determined if we fix the choices (of moves) made during the execution; furthermore, these choices can be specified as a binary string, i.e., a sequence of 0 and 1. Thus, we often identify one computation path with a binary string that defines this computation path. Intuitively, a random bit sequence used by a randomized algorithm corresponds to one of these binary strings.

In general, the output $M(x)$ of the randomized TM M on input x is a random variable, and it may not be what we want; that is, M may make an error. The error probability of M on x is the probability that $M(x)$ does not output the correct answer, including the case that $M(x)$ does not stop. Clearly, a machine is useless if its error probability is large, and thus, for randomized TMs, we consider only machines whose error probability is within some bounds. In particular, for decision problems, the following three types of machines are usually considered. (In the following, let X be any decision problem and let ϵ be any constant <1/2. Recall that we also use X to denote the set of 'yes' instances of the problem X. Each condition is also the definition of the notion "M solves X" for the corresponding machine type. By "$M(x) = ?$" we mean that the machine enters some special halting state indicating 'no answer'.)

Bounded error TM:

for any input x, $x \in X \Rightarrow \Pr_M\{M(x) \neq \text{yes}\} < \epsilon$,

$x \notin X \Rightarrow \Pr_M\{M(x) \neq \text{no}\} < \epsilon$,

One sided error TM:

for any input x, $x \in X \Rightarrow \Pr_M\{M(x) \neq \text{yes}\} < \epsilon$,

$x \notin X \Rightarrow \Pr_M\{M(x) \neq \text{no}\} = 0$.

Zero error TM:

for any input x, $\Pr_M\{M(x) = ?\} < \epsilon$,

$x \in X \Rightarrow \Pr_M\{M(x) = \text{yes or } ?\} = 1$,

$x \notin X \Rightarrow \Pr_M\{M(x) = \text{no or } ?\} = 0$.

The choice of ϵ is not essential in most cases because one can reduce error probability by a constant factor by executing M a constant number of times. The last condition is called "zero error" because one can always get the correct answer by iterating M's execution until a yes/no answer is obtained.

Running time and memory space may also differ depending on an actual computation path executed. We can simply use the worst one for $\text{time}_M(x)$ and $\text{space}_M(x)$.

Nondeterministic Turing Machines (*Nondet. TMs*)

In computational complexity theory, researchers have also introduced machines models that do not have any realistic computational devices. Such machine models are defined for studying some class of problems that share a certain computational property. The *nondeterministic Turing machine* (nondet. TM in short) is a typical example.

As computational devices, nondet. TMs are the same as randomized TMs, but the coin flipping state is called here a *nondeterministic state,* and its interpretation is different. When entering the nondeterministic state, we consider that two moves are executed simultaneously. In other words, in the execution of a nondet. TM, all computation paths are executed *in parallel.* We also require a specification of how to interpret the machine's output from the outputs on these computation paths.

The interpretation is simple for decision problems. (Nondet. TMs are usually used for decision problems.) For any nondet. TM M and for any input x, we consider that M outputs 'yes' on x if $M(x)$ = yes on *some* computation path. Otherwise (i.e., if no computation path yields a yes answer), then we think that M outputs 'no' on x. This way of executing a nondet. TM and interpreting the outputs on computations paths is called *nondeterministic computation.*

There are two important remarks on the nondet. TM model. We should not regard this model as a model for parallel computation. This is because the degree of parallelism, i.e., the number of processors to achieve the parallel computation, easily goes far beyond any realistic level. Furthermore, the nondeterministic computation is a quite limited type of parallel computation. Thus, it is more natural if we consider nondet. TM as a model for defining some classes of problems sharing a certain property (see Section 3). Further, nondeterministic computation is not symmetric. That is, the case that a machine outputs 'yes', and the case that a machine outputs 'no' are not symmetric. Thus, from a nondet. TM M for some problem X, we cannot make a nondet. TM for the complement problem X by simply exchanging its accepting and rejecting states.

Running time and memory space are defined in the same way as randomized TMs. That is, $\text{time}_M(x)$ (resp., $\text{space}_M(x)$) is the longest time (resp., the largest space) used among all computation paths of M on x.

Families of Boolean Circuits (*Circuits*)

Boolean circuits (circuits in short) are used to investigate the running time of parallel algorithms. They are also used as a combinatorial model of computation.

A Boolean circuit is a directed acyclic graph. Nodes of the graph, which are called *gates,* are classified into four types. Nodes with no fan-in edge are *input gates*. The other nodes are either *AND, OR-*, or *NOT-gates.* Nodes with no fan-out edge are also called *output-gates.* We use n to denote the number of input gates. Circuits for decision problems have only one output gate. NOT-gates have only one fan-in edge. On the other hand, AND- or OR-gates may have any number of fan-in edge, but unless explicitly specified, we assume that AND- and OR-gates have two fan-in edges.

A circuit C with n input gates is used to express computation on input strings of length n. That is, for a given string x of length n, $C(x)$, the output of C on input x, is the value obtained at its output gate when the circuit is evaluated by assigning the *i*-th bit of x to the *i*-th input gate. Since each circuit can take care of strings of some fixed length, the whole computation is represented by a family $\{C_n\}_{n\geq 0}$ of circuits, where each C_n is a circuit with n input gates. If there is no restriction, then there need be no regularity between the circuits in $\{C_n\}_{n\geq 0}$, and then there may be no way to construct C_n from n. Due to this, a family of circuits is generally called a *nonuniform computation model.* On the other hand, we can restrict circuit families so that each member is constructible from a given size by some resource-bounded det. T.M. Such circuit families are called *uniform families of circuits.* For example, in order to give a relation to actual parallel computing devices, log-space uniformity is often assumed. But in many cases, the uniformity issue is not explicitly stated because it is either not necessary or easily obtained in the argument. In this chapter, we will also omit mentioning uniformity.

For the circuit model, we measure computational resource by "depth" and "size". The *depth* of a circuit C, which we denote as $\text{depth}(C)$, is the length of the longest path from an input gate to the output gate of C. The size of C, $\text{size}(C)$, is the total number of AND-, OR-, and NOT-gates. It is intuitively clear (and in fact, probable in some settings) that circuit depth corresponds to parallel running time. On the other hand, circuit size corresponds to the combination of the time and space resources of sequential algorithms.

We have seen several machine models and resources defined on them. Now we explain the way to evaluate a given algorithm's performance as a total. Here for our example, we consider a deterministic algorithm A and time $_A(x)$.

As explained in the introduction, we evaluate the efficiency of algorithms in terms of input length. But even if we fix input length n, there are many input strings of length n, and A 's running time usually varies depending on an actual input string. In computational complexity theory, we usually choose the worst-case criteria and define the *worst-case time complexity* of A as follows.

$$\text{time}_A(n) \;=\; \max\{\text{time}_A(x) \mid x \in \{0,1\}^n\}.$$

A function like time $_A(n)$ is called a *time complexity* (*function*). Clearly, the worst-case criterion is not our only choice, and it is not always the best choice either. For example, there are cases that the average-case analysis is more appropriate. But since the average-case analysis is usually complex, and it is even more difficult to provide a simple and appropriate framework for studying the average-case complexity, average-case complexity has not been studied until recently (see Section 5).

In complexity theory and analysis of algorithms, researchers have mainly studied the asymptotic behavior of complexity functions, by ignoring constant factors and minor

terms of complexity function. This is because even asymptotic complexity is hard enough to analyze, and furthermore, with our rough machine models it is often meaningless to discuss constant factors of complexity functions. Thus, for stating complexity functions, we usually use *big-O notation*, the way to describe functions' asymptotic behavior while ignoring constant factors. Here we recall the meaning of the big-O notation.

Definition 1 *For any functions* f *and* g *from* $\mathbf{Z}^+$ *to* $\mathbf{Z}^+$, *we write* $f = O(g)$ *if the following holds for some constant* c_0 *and* n_0 $\forall n \geq n_0[\, f(n) \leq c_0 \cdot g(n)]$.

Note that the big-O notation does not imply the asymptotic equivalence between two functions; it just means that one function is asymptotically bounded *from above* by another function. Thus, we can write $f = O(g)$, for example, for $f(n) = n$ and $g(n) = 2^n$.

-
-
-

Bibliography

[1] Balcázar, L.J., Díaz, J., Gabarró, J. (1988), S*tructural Complexity I*, 191 pp. Berlin: Springer-Verlag. [A standard text book on complexity theory, which can be used as a reference book for basic notions and notations.]

[2] Garey, M.R., Johnson, D.S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*, 340 pp. San Francisco: Freeman. [A comprehensive text book on NP problems with a list of more than 300 NP complete problems.]

[3] Van Leeuwen J., Ed. (1990). *Handbook of Theoretical Computer Science: Volume A, Al-gorithms and Complexity*, 996 pp. Cambridge, MA: MIT Press. [A collection of surveys on major areas concerning algorithms and complexity theory.]

[4] Sipser, M. (1997). *Introduction to the Theory of Computation*, 396 pp. Boston, MA:PWS Publishing Co. [A basic text book on theory of computing.]

[5] Stearns, R. (1990). Juris Hartmanis: The beginning of computational complexity, in *Complexity Theory Retrospective* (ed. A. Selman), 5-18. Berlin: Springer-Verlag. [A survey on Hierarchy theorems of Stearns and Hartmanis.]

Biographical Sketch

Osamu Watanabe was born in March 1958. He received BS, MS, and Dr. Ing. Degrees respectively in 1980, 1982, and 1987, all from Tokyo Institute of Technology**.** He has been with Tokyo Institute of Technology since 1982, a professor since 1997.