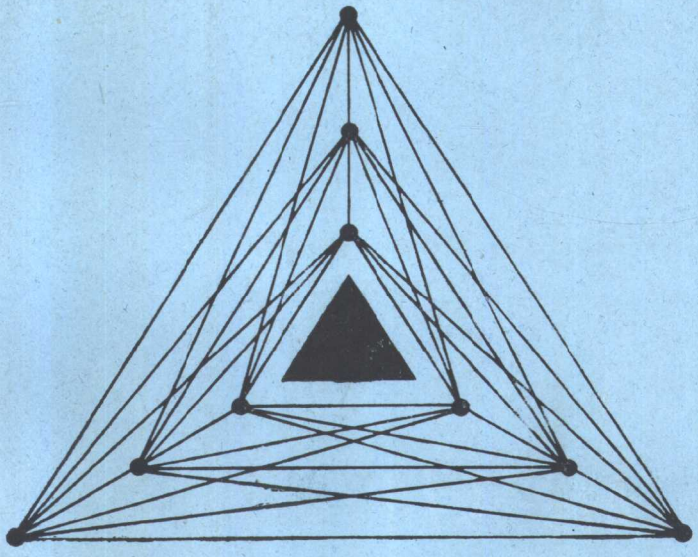
COMPUTERS AND INTRACTABILITY

A Guide to the Theory of NP-Completeness

Michael R. Garey / David S. Johnson



COMPUTERS AND INTRACTABILITY

A Guide to the Theory of NP-Completeness

Michael R. Garey / David S. Johnson

BELL LABORATORIES
MURRAY HILL, NEW JERSEY

W. H. FREEMAN AND COMPANY
San Francisco

Library of Congress Cataloging in Publication Data

Garey, Michael R.
Computers and Intractability.

Bibliography: p.
Includes index.
1. Electronic digital computers--Programming.
2. Algorithms. 3. Computational complexity.
I. Johnson, David S., joint author. II. Title.
III. Title: NP-completeness.
QA76.6.G35 519.4 78-12361
ISBN 0-7167-1044-7
ISBN 0-7167-1045-5 pbk.

AMS Classification: Primary 68A20
Computer Science: Computational complexity and efficiency

Printed in the United States of America

9 8 7 6 5 4 3 2 1

Preface

Few technical terms have gained such rapid notoriety as the appellation "NP-complete." In the short time since its introduction in the early 1970's, this term has come to symbolize the abyss of inherent intractability that algorithm designers increasingly face as they seek to solve larger and more complex problems. A wide variety of commonly encountered problems from mathematics, computer science, and operations research are now known to be NP-complete, and the collection of such problems continues to grow almost daily. Indeed, the NP-complete problems are now so pervasive that it is important for anyone concerned with the computational aspects of these fields to be familiar with the meaning and implications of this concept.

This book is intended as a detailed guide to the theory of NP-completeness, emphasizing those concepts and techniques that seem to be most useful for applying the theory to practical problems. It can be viewed as consisting of three parts.

The first part, Chapters 1 through 5, covers the basic theory of NP-completeness. Chapter 1 presents a relatively low-level introduction to some of the central notions of computational complexity and discusses the significance of NP-completeness in this context. Chapters 2 through 5 provide the detailed definitions and proof techniques necessary for thoroughly understanding and applying the theory.

The second part, Chapters 6 and 7, provides an overview of two alternative directions for further study. Chapter 6 concentrates on the search for efficient "approximation" algorithms for NP-complete problems, an area whose development has seen considerable interplay with the theory of NP-completeness. Chapter 7 surveys a large number of theoretical topics in computational complexity, many of which have arisen as a consequence of previous work on NP-completeness. Both of these chapters (especially Chapter 7) are intended solely as introductions to these areas, with our expectation being that any reader wishing to pursue particular topics in more detail will do so by consulting the cited references.

The third and final part of the book is the Appendix, which contains an extensive list (more than 300 main entries, and several times this many results in total) of NP-complete and NP-hard problems. Annotations to the main entries discuss what is known about the complexity of subproblems and variants of the stated problems.

The book should be suitable for use as a supplementary text in courses on algorithm design, computational complexity, operations research, or combinatorial mathematics. It also can be used as a starting point for seminars on approximation algorithms or computational complexity at the graduate or advanced undergraduate level. The second author has used a preliminary draft as the basis for a graduate seminar on approximation algorithms, covering Chapters 1 through 5 in about five weeks and then pursuing the topics in Chapter 6, supplementing them extensively with additional material from the references. A seminar on computational complexity might proceed similarly, substituting Chapter 7 for Chapter 6 as the initial access point to the literature. It is also possible to cover both chapters in a combined seminar.

More generally, the book can serve both as a self-study text for anyone interested in learning about the subject of NP-completeness and as a reference book for researchers and practitioners who are concerned with algorithms and their complexity. The list of NP-complete problems in the Appendix can be used by anyone familiar with the central notions of NP-completeness, even without having read the material in the main text. The novice can gain such familiarity by skimming the material in Chapters 1 through 5, concentrating on the informal discussions of definitions and techniques, and returning to the more formal material only as needed for clarification. To aid those using the book as a reference, we have included a substantial number of terms in the Subject Index, and the extensive Reference and Author Index gives the sections where each reference is mentioned in the text.

We are indebted to a large number of people who have helped us greatly in preparing this book. Hal Gabow, Larry Landweber, and Bob Tarjan taught from preliminary versions of the book and provided us with valuable suggestions based on their experience. The following people read preliminary drafts of all or part of the book and made constructive comments: Al Aho, Shimon Even, Ron Graham, Harry Hunt, Victor Klee, Albert Meyer, Christos Papadimitriou, Henry Pollak, Sartaj Sahni, Ravi Sethi, Larry Stockmeyer, and Jeff Ullman. A large number of researchers, too numerous to mention here (but see the Reference and Author Index), responded to our call for NP-completeness results and contributed toward making our list of NP-complete problems as extensive as it is. Several of our colleagues at Bell Laboratories, especially Brian Kernighan, provided invaluable assistance with computer typesetting on the UNIX® system. Finally, special thanks go to Jeanette Reinbold, whose facility with translating our handwritten hieroglyphics into faultless input to the typesetting system made the task of writing this book so much easier.

Murray Hill, New Jersey
October, 1978

MICHAEL R. GAREY
DAVID S. JOHNSON

Contents

1

Computers, Complexity, and Intractability

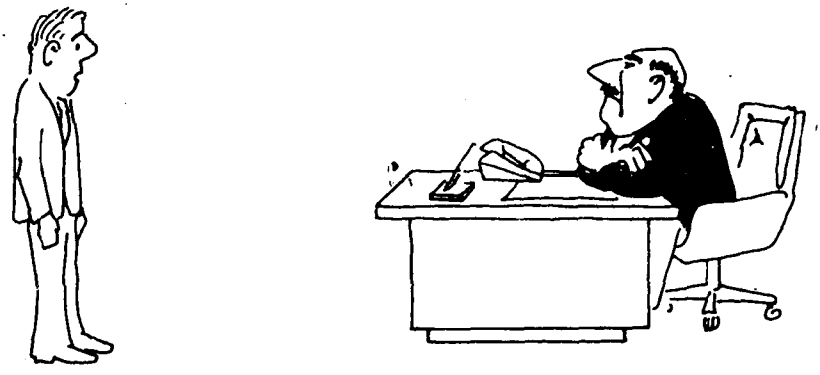
1.1 Introduction

The subject matter of this book is perhaps best introduced through the following, somewhat whimsical, example.

Suppose that you, like the authors, are employed in the halls of industry. One day your boss calls you into his office and confides that the company is about to enter the highly competitive "bandersnatch" market. For this reason, a good method is needed for determining whether or not any given set of specifications for a new bandersnatch component can be met and, if so, for constructing a design that meets them. Since you are the company's chief algorithm designer, your charge is to find an efficient algorithm for doing this.

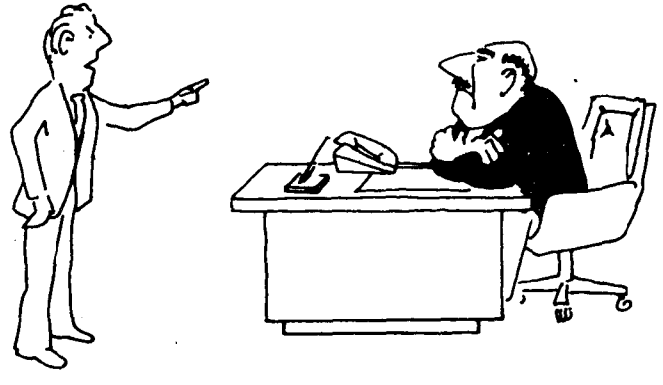
After consulting with the bandersnatch department to determine exactly what the problem is, you eagerly hurry back to your office, pull down your reference books, and plunge into the task with great enthusiasm. Some weeks later, your office filled with mountains of crumpled-up scratch paper, your enthusiasm has lessened considerably. So far you have not been able to come up with any algorithm substantially better than searching through all possible designs. This would not particularly endear you to your boss, since it would involve years of computation time for just one set of

specifications, and the bandersnatch department is already 13 components behind schedule. You certainly don't want to return to his office and report:



"I can't find an efficient algorithm, I guess I'm just too dumb."

To avoid serious damage to your position within the company, it would be much better if you could prove that the bandersnatch problem is *inherently* intractable, that no algorithm could possibly solve it quickly. You then could stride confidently into the boss's office and proclaim:



"I can't find an efficient algorithm, because no such algorithm is possible!"

Unfortunately, proving inherent intractability can be just as hard as finding efficient algorithms. Even the best theoreticians have been stymied in their attempts to obtain such proofs for commonly encountered hard problems. However, having read this book, you have discovered something

almost as good. The theory of NP-completeness provides many straightforward techniques for proving that a given problem is "just as hard" as a large number of other problems that are widely recognized as being difficult and that have been confounding the experts for years. Armed with these techniques, you might be able to prove that the bandersnatch problem is NP-complete and, hence, that it is equivalent to all these other hard problems. Then you could march into your boss's office and announce:

"I can't find an efficient algorithm, but neither can all these famous people."

At the very least, this would inform your boss that it would do no good to fire you and hire another expert on algorithms.

Of course, our own bosses would frown upon our writing this book if its sole purpose was to protect the jobs of algorithm designers. Indeed, discovering that a problem is NP-complete is usually just the beginning of work on that problem. The needs of the bandersnatch department won't disappear overnight simply because their problem is known to be NP-complete. However, the knowledge that it is NP-complete does provide valuable information about what lines of approach have the potential of being most productive. Certainly the search for an efficient, exact algorithm should be accorded low priority. It is now more appropriate to concentrate on other, less ambitious, approaches. For example, you might look for efficient algorithms that solve various special cases of the general problem. You might look for algorithms that, though not guaranteed to run quickly, seem likely to do so most of the time. Or you might even relax the problem somewhat, looking for a fast algorithm that merely finds designs that

meet *most* of the component specifications. In short, the primary application of the theory of NP-completeness is to assist algorithm designers in directing their problem-solving efforts toward those approaches that have the greatest likelihood of leading to useful algorithms.

In the first chapter of this "guide" to NP-completeness, we introduce many of the underlying concepts, discuss their applicability (as well as give some cautions), and outline the remainder of the book.

1.2 Problems, Algorithms, and Complexity

In order to elaborate on what is meant by "inherently intractable" problems and problems having "equivalent" difficulty, it is important that we first agree on the meaning of several more basic terms.

Let us begin with the notion of a problem. For our purposes, a *problem* will be a general question to be answered, usually possessing several *parameters,* or free variables, whose values are left unspecified. A problem is described by giving: (1) a general description of all its parameters, and (2) a statement of what properties the answer, or *solution,* is required to satisfy. An *instance* of a problem is obtained by specifying particular values for all the problem parameters.

As an example, consider the classical "traveling salesman problem." The parameters of this problem consist of a finite set $C = \{c_1, c_2, \ldots, c_m\}$ of "cities" and, for each pair of cities c_i, c_j in C, the "distance" $d(c_i, c_j)$ between them. A solution is an ordering $\langle c_{\pi(1)}, c_{\pi(2)}, \ldots, c_{\pi(m)} \rangle$ of the given cities that minimizes

$$\left[\sum_{i=1}^{m-1} d(c_{\pi(i)}, c_{\pi(i+1)})\right] + d(c_{\pi(m)}, c_{\pi(1)})$$

This expression gives the length of the "tour" that starts at $c_{\pi(1)}$, visits each city in sequence, and then returns directly to $c_{\pi(1)}$ from the last city $c_{\pi(m)}$.

One instance of the traveling salesman problem, illustrated in Figure 1.1, is given by $C = \{c_1, c_2, c_3, c_4\}$, $d(c_1, c_2) = 10$, $d(c_1, c_3) = 5$, $d(c_1, c_4) = 9$, $d(c_2, c_3) = 6$, $d(c_2, c_4) = 9$, and $d(c_3, c_4) = 3$. The ordering $\langle c_1, c_2, c_4, c_3 \rangle$ is a solution for this instance, as the corresponding tour has the minimum possible tour length of 27.

Algorithms are general, step-by-step procedures for solving problems. For concreteness, we can think of them simply as being computer programs, written in some precise computer language. An algorithm is said to *solve* a problem Π if that algorithm can be applied to any instance I of Π and is guaranteed always to produce a solution for that instance I. We emphasize that the term "solution" is intended here strictly in the sense introduced above, so that, in particular, an algorithm does not "solve" the traveling

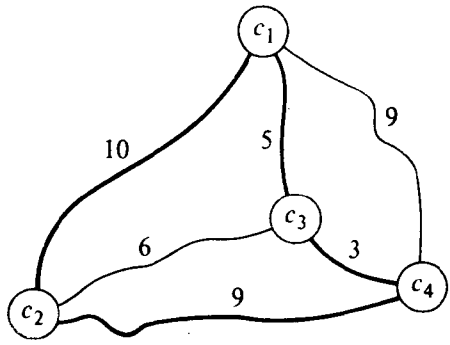


Figure 1.1 An instance of the traveling salesman problem and a tour of length 27, which is the minimum possible in this case.

salesman problem unless it always constructs an ordering that gives a minimum length tour.

In general, we are interested in finding the most "efficient" algorithm for solving a problem. In its broadest sense, the notion of efficiency involves all the various computing resources needed for executing an algorithm. However, by the "most efficient" algorithm one normally means the fastest. Since time requirements are often a dominant factor determining whether or not a particular algorithm is efficient enough to be useful in practice, we shall concentrate primarily on this single resource.

The time requirements of an algorithm are conveniently expressed in terms of a single variable, the "size" of a problem instance, which is intended to reflect the amount of input data needed to describe the instance. This is convenient because we would expect the relative difficulty of problem instances to vary roughly with their size. Often the size of a problem instance is measured in an informal way. For the traveling salesman problem, for example, the number of cities is commonly used for this purpose. However, an m-city problem instance includes, in addition to the labels of the m cities, a collection of $m(m-1)/2$ numbers defining the inter-city distances, and the sizes of these numbers also contribute to the amount of input data. If we are to deal with time requirements in a precise, mathematical manner, we must take care to define instance size in such a way that all these factors are taken into account.

To do this, observe that the description of a problem instance that we provide as input to the computer can be viewed as a single finite string of symbols chosen from a finite input alphabet. Although there are many different ways in which instances of a given problem might be described, let us assume that one particular way has been chosen in advance and that each problem has associated with it a fixed *encoding scheme*, which maps problem

instances into the strings describing them. The *input length* for an instance I of a problem Π is defined to be the number of symbols in the description of I obtained from the encoding scheme for Π. It is this number, the input length, that is used as the formal measure of instance size.

For example, instances of the traveling salesman problem might be described using the alphabet $\{c, [,], /, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, with our previous example of a problem instance being encoded by the string "$c[1]c[2]c[3]c[4]//10/5/9//6/9//3$." More complicated instances would be encoded in analogous fashion. If this were the encoding scheme associated with the traveling salesman problem, then the input length for our example would be 32.

The *time complexity function* for an algorithm expresses its time requirements by giving, for each possible input length, the largest amount of time needed by the algorithm to solve a problem instance of that size. Of course, this function is not well-defined until one fixes the encoding scheme to be used for determining input length and the computer or computer model to be used for determining execution time. However, as we shall see, the particular choices made for these will have little effect on the broad distinctions made in the theory of NP-completeness. Hence, in what follows, the reader is advised merely to fix in mind a particular encoding scheme for each problem and a particular computer or computer model, and to think in terms of time complexity as determined from the corresponding input lengths and execution times.

1.3 Polynomial-Time Algorithms and Intractable Problems

Different algorithms possess a wide variety of different time complexity functions, and the characterization of which of these are "efficient enough" and which are "too inefficient" will always depend on the situation at hand. However, computer scientists recognize a simple distinction that offers considerable insight into these matters. This is the distinction between polynomial time algorithms and exponential time algorithms.

Let us say that a function $f(n)$ is $O(g(n))$ whenever there exists a constant c such that $|f(n)| \leqslant c \cdot |g(n)|$ for all values of $n \geqslant 0$. A *polynomial time algorithm* is defined to be one whose time complexity function is $O(p(n))$ for some polynomial function p, where n is used to denote the input length. Any algorithm whose time complexity function cannot be so bounded is called an *exponential time algorithm* (although it should be noted that this definition includes certain non-polynomial time complexity functions, like $n^{\log n}$, which are not normally regarded as exponential functions).

The distinction between these two types of algorithms has particular significance when considering the solution of large problem instances. Figure 1.2 illustrates the differences in growth rates among several typical complexity functions of each type, where the functions express execution time

in terms of microseconds. Notice the much more explosive growth rates for the two exponential complexity functions.

Time complexity function	Size n					
	10	20	30	40	50	60
n	.00001 second	.00002 second	.00003 second	.00004 second	.00005 second	.00006 second
n^2	.0001 second	.0004 second	.0009 second	.0016 second	.0025 second	.0036 second
n^3	.001 second	.008 second	.027 second	.064 second	.125 second	.216 second
n^5	.1 second	3.2 seconds	24.3 seconds	1.7 minutes	5.2 minutes	13.0 minutes
2^n	.001 second	1.0 second	17.9 minutes	12.7 days	35.7 years	366 centuries
3^n	.059 second	58 minutes	6.5 years	3855 centuries	2×10^8 centuries	1.3×10^{13} centuries

Figure 1.2 Comparison of several polynomial and exponential time complexity functions.

Even more revealing is an examination of the effects of improved computer technology on algorithms having these time complexity functions. Figure 1.3 shows how the largest problem instance solvable in one hour would change if we had a computer 100 or 1000 times faster than our present machine. Observe that with the 2^n algorithm a thousand-fold increase in computing speed only adds 10 to the size of the largest problem instance we can solve in an hour, whereas with the n^5 algorithm this size almost quadruples.

These tables indicate some of the reasons why polynomial time algorithms are generally regarded as being much more desirable than exponential time algorithms. This view, and the distinction between the two types of algorithms, is central to our notion of inherent intractability and to the theory of NP-completeness.

The fundamental nature of this distinction was first discussed in [Cobham, 1964] and [Edmonds, 1965a]. Edmonds, in particular, equated poly-

Size of Largest Problem Instance Solvable in 1 Hour

Time complexity function	With present computer	With computer 100 times faster	With computer 1000 times faster
n	N_1	$100\ N_1$	$1000\ N_1$
n^2	N_2	$10\ N_2$	$31.6\ N_2$
n^3	N_3	$4.64\ N_3$	$10\ N_3$
n^5	N_4	$2.5\ N_4$	$3.98\ N_4$
2^n	N_5	$N_5+6.64$	$N_5+9.97$
3^n	N_6	$N_6+4.19$	$N_6+6.29$

Figure 1.3 Effect of improved technology on several polynomial and exponential time algorithms.

nomial time algorithms with "good" algorithms and conjectured that certain integer programming problems might not be solvable by such "good" algorithms. This reflects the viewpoint that exponential time algorithms should not be considered "good" algorithms, and indeed this usually is the case. Most exponential time algorithms are merely variations on exhaustive search, whereas polynomial time algorithms generally are made possible only through the gain of some deeper insight into the structure of a problem. There is wide agreement that a problem has not been "well-solved" until a polynomial time algorithm is known for it. Hence, we shall refer to a problem as *intractable* if it is so hard that no polynomial time algorithm can possibly solve it.

Of course, this formal use of "intractable" should be viewed only as a rough approximation to its dictionary meaning. The distinction between "efficient" polynomial time algorithms and "inefficient" exponential time algorithms admits of many exceptions when the problem instances of interest have limited size. Even in Figure 1.2, the 2^n algorithm is faster than the n^5 algorithm for $n \leqslant 20$. More extreme examples can be constructed easily.

Furthermore, there are some exponential time algorithms that have been quite useful in practice. Time complexity as defined is a *worst-case* measure, and the fact that an algorithm has time complexity 2^n means only that at least one problem instance of size n requires that much time. Most problem instances might actually require far less time than that, a situation

that appears to hold for several well-known algorithms. The simplex algorithm for linear programming has been shown to have exponential time complexity [Klee and Minty, 1972], [Zadeh, 1973], but it has an impressive record of running quickly in practice. Likewise, branch-and-bound algorithms for the knapsack problem have been so successful that many consider it to be a "well-solved" problem, even though these algorithms, too, have exponential time complexity.

Unfortunately, examples like these are quite rare. Although exponential time algorithms are known for many problems, few of them are regarded as being very useful in practice. Even the successful exponential time algorithms mentioned above have not stopped researchers from continuing to search for polynomial time algorithms for solving those problems. In fact, the very success of these algorithms has led to the suspicion that they somehow capture a crucial property of the problems whose refinement could lead to still better methods. So far, little progress has been made toward explaining this success, and no methods are known for predicting in advance that a given exponential time algorithm will run quickly in practice.

On the other hand, the much more stringent bounds on execution time satisfied by polynomial time algorithms often permit such predictions to be made. Even though an algorithm having time complexity n^{100} or $10^{99}n^2$ might not be considered likely to run quickly in practice, the polynomially solvable problems that arise naturally tend to be solvable within polynomial time bounds that have degree 2 or 3 at worst and that do not involve extremely large coefficients. Algorithms satisfying such bounds *can* be considered to be "provably efficient," and it is this much-desired property that makes polynomial time algorithms the preferred way to solve problems.

Our definition of "intractable" also provides a theoretical framework of considerable generality and power. The intractability of a problem turns out to be essentially independent of the particular encoding scheme and computer model used for determining time complexity.

Let us first consider encoding schemes. Suppose for example that we are dealing with a problem in which each instance is a graph $G = (V,E)$, where V is the set of vertices and E is the set of edges, each edge being an unordered pair of vertices. Such an instance might be described (see Figure 1.4) by simply listing all the vertices and edges, or by listing the rows of the adjacency matrix for the graph, or by listing for each vertex all the other vertices sharing a common edge with it (a "neighbor" list). Each of these encodings can give a different input length for the same graph. However, it is easy to verify (see Figure 1.5) that the input lengths they determine differ at most polynomially from one another, so that any algorithm having polynomial time complexity under one of these encoding schemes also will have polynomial time complexity under all the others. In fact, the standard encoding schemes used in practice for any particular problem always seem to differ at most polynomially from one another. It would be difficult to imagine a "reasonable" encoding scheme for a problem that differs more