


Digitized by the Internet Archive
in 2022 with funding from
Kahle/Austin Foundation

https://archive.org/details/computationalcom0000papa

COMPUTATIONAL COMPLEXITY

Christos H. Papadimitriou

University of California – San Diego



Addison-Wesley Publishing Company

Reading, Massachusetts • Menlo Park, California • New York
Don Mills, Ontario • Wokingham, England • Amsterdam • Bonn • Sydney
Singapore • Tokyo • Madrid • San Juan • Milan • Paris

Cover Photo: Courtesy of Erich Lessing, Art Resource, NY.

Library of Congress Cataloging-in-Publication Data

Papadimitriou, Christos M.
Computational complexity / by Christos H. Papadimitriou
p. cm.
Includes bibliographical references and index.
ISBN 0-201-53082-1
1. Computational complexity. I. Title.
QA267.7.P36 1994
511.3–dc20 93-5662
CIP

The procedures and applications presented in this book have been included for their instructional value. They have been tested with care but are not guaranteed for any particular purpose. The publisher does not offer any warranties or representations, nor does it accept any liabilities with respect to the programs or applications.

Copyright © 1994 by Addison-Wesley Publishing Company, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America.

1 2 3 4 5 6 7 8 9 10-MA-959493

To Victoria Papadimitriou (1910–1992)
for the complexity that she gave me

Preface

I wish nothing else but to speak simply
please grant me this privilege
because we have burdened our song with so much music
that it is slowly sinking
and our art has become so ornate
that the makeup has corroded her face
and it is time to say our few simple words
because tomorrow our soul sails away

Giorgos Seferis

This book is an introduction to the theory of computational complexity at a level appropriate for a beginning graduate or advanced undergraduate course. Computational complexity is the area of computer science that contemplates the reasons why some problems are so hard to solve by computers. This field, virtually non-existent only 20 years ago, has expanded tremendously and now comprises a major part of the research activity in theoretical computer science. No book on complexity can be comprehensive now—certainly this one is not. *It only contains results which I felt I could present clearly and relatively simply, and which I consider central to my point of view of complexity.*

At the risk of burdening the reader so early with a message that will be heard rather frequently and loudly throughout the book's twenty chapters, my point of view is this: I see complexity as the intricate and exquisite interplay between *computation* (complexity classes) and *applications* (that is, problems). Completeness results are obviously central to this approach. So is logic, a most important application that happens to excel in expressing and capturing computation. Computation, problems, and logic are thus the three main currents that run through the book.

Contents

For a quick look at the table of contents, Chapter 1 introduces problems and algorithms—because complexity is best understood when contrasted with simplicity. Chapter 2 treats Turing machines, while at the same time making the point that our approach is very much machine-independent. Chapter 3 is an introduction to undecidability (not only the highest form of complexity, but also a major methodological influence).

Next comes logic. I am aware that this is the part that will seem most alien and unusual to my fellow complexity theorists. But it is so relevant to my way of viewing complexity, so basic for computer science in general, and so rarely treated in a way accessible to computer scientists, that I felt I had to give it a try. Chapter 4 is about Boolean logic (including the algorithmic properties of Horn clauses, and circuits up to Shannon's theorem). Then in Chapter 5 comes first-order logic, its model theory and its proof theory up to the completeness theorem, and enough second-order logic to usher in later Fagin's characterization of NP—a very useful and often overlooked parallel of Cook's theorem. Chapter 6 is a self-contained proof of Gödel's incompleteness theorem, an important early instance of logic expressing computation.

Then complexity is taken on in earnest. Chapter 7 is an exposition of the known relations between complexity classes —including the theorems by Savitch and Immerman-Szelepscényi on space complexity. In Chapter 8 is an introduction to reductions and the concept of completeness, immediately exemplified by Cook's theorem and the P-completeness of the circuit value problem; the parallel of the characterizations of P and NP in terms of logic is also pursued here. Chapter 9 contains many NP-completeness results, sprinkled with tips on proof methodology. Chapter 10 discusses coNP and function problems, while Chapter 11 introduces randomized algorithms, the complexity classes they define, and their implementation in terms of realistic random sources. Circuits and their relationship with complexity and randomization are also introduced there. Chapter 12 is a brief and rather informal introduction to the important subject of cryptography and protocols. Chapter 13 discusses approximation algorithms, and the recent impossibility results based on probabilistically checkable proofs. On the other hand, Chapter 14 surveys "structural" aspects of the $P \stackrel{?}{=} NP$ question, such as intermediate degrees, isomorphism and density, and oracles; it also contains a proof of Razborov's lower bound for monotone circuits.

For a more careful look inside P, parallel algorithms and complexity are the subject of Chapter 15, while Chapter 16 concentrates on logarithmic space, including the random walk algorithm for undirected paths. Finally, beyond NP we find the polynomial hierarchy in Chapter 17 (with Krentel's characterization of optimization problems); counting problems and Valiant's theorem on permanents in Chapter 18; and the many facets of polynomial space (not the

least interesting of which is Shamir's recent theorem on interactive protocols) in Chapter 19. The book ends with a glimpse to the intractable territory beyond that.

There are no real mathematical prerequisites —besides of course a certain level of "mathematical maturity," a term always left safely undefined in prefaces. All theorems are proved from first principles (with the exception of two theorems that are quoted without proofs in Chapter 13 on approximability), while many more relevant results are stated in the "notes and problems" sections. The proofs and constructions are often much simpler than one finds in the literature. In fact, the book contains brief and gentle introductions to several subjects as they relate to complexity: Enough elementary number theory to prove Pratt's theorem, the Solovay-Strassen primality test, and the RSA cryptographic protocol (Chapters 10, 11, and 12); elementary probability (Chapters 11 and elsewhere); combinatorics and the probabilistic method (Chapters 10, 13, and 14); recursion theory (Chapters 3 and 14); and, of course, logic (Chapters 4, 5, and 6). Since complexity questions always follow a reasonably comprehensive development of the corresponding algorithmic ideas (efficient algorithms in Chapter 1, randomized algorithms in Chapter 11, approximation algorithms in Chapter 13, and parallel algorithms in Chapter 15), the book is also a passable introduction to algorithms —although only rough analysis, enough to establish polynomiality, is attempted in each case.

Notes and Problems

Each chapter ends with a section that contains intertwined references, notes, exercises, and problems; many of the problems are hand-held tours of further results and topics. In my view this is perhaps the most important section of the chapter (it is often by far the longest one), and you should consider reading it as part of the text. It usually gives historical perspective and places the chapter within the broader field. Most problems are followed with a parenthetical hint and/or references. All are doable, at least following the hint or looking up the solution in the library (I have found that this is often at least as valuable to my students as passing yet another IQ test). There is no difficulty code for problems, but you are warned about any really hard ones.

Teaching

For all its obvious emphasis on complexity, this book has been designed (and used) as a general introduction to the theory of computation for computer scientists. My colleagues and I have used it over the past three years in a ten-week course mainly intended for first-year computer science masters students at the University of California at San Diego. Two weeks are usually enough for a

review of the first four chapters, generally familiar from the students' undergraduate training. Logic takes the next three weeks, often without the proof of the completeness theorem. The remaining five weeks are enough for Chapter 7, a serious drill on NP-completeness (not included in algorithms courses at UCSD), and a selection of one or two among Chapters 11 through 14. A semester-length course could cover all four of these chapters. If you must skip logic, then Chapter 15 on parallelism can be added (however, I believe that a good part of the book's message and value would be lost this way).

There are at least two other courses one can teach from this book: The subjects in the first nine chapters are, in my view, so central for computer scientists, that they can replace with pride the usual automata and formal languages in an advanced undergraduate introductory theory course (especially since many compiler courses are nowadays self-contained in this respect). Also, I have used twice the last eleven chapters for a second course in theory; the goal here is to bring interested graduate students up to the research issues in complexity—or, at least, help them be an informed audience at a theory conference.

Debts

My thinking on complexity was shaped by a long process of exciting and inspiring interactions with my teachers, students, and colleagues (especially those who were all three). I am immensely grateful to all of them: Ken Steiglitz, Jeff Ullman, Dick Karp, Harry Lewis, John Tsitsiklis, Don Knuth, Steve Vavasis, Jack Edmonds, Albert Meyer, Gary Miller, Patrick Dymond, Paris Kanellakis, David Johnson, Elias Koutsoupias (who also helped me a lot with the figures, the final check, and the index), Umesh Vazirani, Ken Arrow, Russell Impagliazzo, Sam Buss, Milena Mihail, Vijay Vazirani, Paul Spirakis, Pierluigi Crescenzi, Noga Alon, Stathis Zachos, Heather Woll, Phokion Kolaitis, Neil Immerman, Pete Veinott, Joan Feigenbaum, Lefteris Kirousis, Deng Xiaotie, Foto Afrati, Richard Anderson; and above all, Mihalis Yannakakis and Mike Sipser. Many of them read early drafts of this book and contributed criticisms, ideas, and corrections or made me nervous with their silence. Of all the students in the course who commented on my lecture notes I can only remember the names of David Morgenthaler, Goran Gogic, and Markus Jacobsson (but I do remember the smiles of the rest). Naturally, I am responsible for the remaining errors– although, in my opinion, my friends could have caught a few more.

I am very grateful to Martha Sideri for her sweet encouragement and support, as well as her notes, ideas, opinions, and help with the cover design.

I worked on writing this book at the University of California, San Diego, but also during visits at AT&T Bell Laboratories, the University of Bonn, the Max-Planck-Institut at Saarbrücken, the University of Patras and the Com-

puter Technology Institute there, and the Université de Paris Sud. My research on algorithms and complexity was supported by the National Science Foundation, the Esprit project ALCOM, and the Irwin Mark and Joan Klein Jacobs chair for information and computer sciences at the University of California at San Diego.

It was a pleasure to work on this project with Tom Stone and his colleagues at Addison-Wesley. Finally, I typeset the book using Don Knuth's TEX, and my macros evolved from those Jeff Ullman gave me many years ago.

Christos H. Papadimitriou
La Jolla, summer of 1993

Contents

PART I: ALGORITHMS 1

1 **Problems and Algorithms** 3

1.1 Graph reachability 3
1.2 Maximum flow and matching 8
1.3 The traveling salesman problem 13
1.4 Notes, references, and problems 14

2 **Turing machines** 19

2.1 Turing machine basics 19
2.2 Turing machines as algorithms 24
2.3 Turing machines with multiple strings 26
2.4 Linear speedup 32
2.5 Space bounds 34
2.6 Random access machines 36
2.7 Nondeterministic machines 45
2.8 Notes, references, and problems 51

3 **Computability** 57

3.1 Universal Turing machines 57
3.2 The halting problem 58
3.3 More undecidability 60
3.4 Notes, references, and problems 66

PART II: LOGIC 71

4 Boolean logic 73

4.1 Boolean Expressions
4.2 Satisfiability and validity 76
4.3 Boolean functions and circuits 79
4.4 Notes, references, and problems 84

5 First-order logic 87

5.1 The syntax of first-order logic 87
5.2 Models 90
5.3 Valid expressions 95
5.4 Axioms and proofs 100
5.5 The completeness theorem 105
5.6 Consequences of the completeness theorem 110
5.7 Second-order logic 113
5.8 Notes, references, and problems 118

6 Undecidability in logic 123

6.1 Axioms for number theory 123
6.2 Complexity as a number-theoretic concept 127
6.3 Undecidability and incompleteness 131
6.4 Notes, references, and problems 135

PART III: P AND NP 137

7 Relations between complexity classes 139

7.1 Complexity classes 139
7.2 The hierarchy theorem 143
7.3 The reachability method 146
7.4 Notes, references, and problems 154

8 Reductions and completeness 159

8.1 Reductions 159
8.2 Completeness 165

8.3 Logical characterizations 172
8.4 Notes, references, and problems 177

9 NP-complete problems **181**

9.1 Problems in **NP** 181
9.2 Variants of satisfiability 183
9.3 Graph-theoretic problems 188
9.4 Sets and numbers 199
9.5 Notes, references, and problems 207

10 coNP and function problems **219**

10.1 **NP** and **coNP** 219
10.2 Primality 222
10.3 Function problems 227
10.4 Notes, references, and problems 235

11 Randomized computation **241**

11.1 Randomized algorithms 241
11.2 Randomized complexity classes 253
11.3 Random sources 259
11.4 Circuit complexity 267
11.5 Notes, references, and problems 272

12 Cryptography **279**

12.1 One-way functions 279
12.2 Protocols 287
12.3 Notes, references, and problems 294

13 Approximability **299**

13.1 Approximation algorithms 299
13.2 Approximation and complexity 309
13.3 Nonapproximability 319
13.4 Notes, references, and problems 323

14 On P vs. NP **329**

14.1 The map of **NP** 329
14.2 Isomorphism and density 332
14.3 Oracles 339
14.4 Monotone circuits 343
14.5 Notes, references, and problems 350

PART IV: INSIDE P **357**

15 Parallel computation **359**

15.1 Parallel algorithms 359
15.2 Parallel models of computation 369
15.3 The class **NC** 375
15.4 **RNC** algorithms 381
15.5 Notes, references, and problems 385

16 Logarithmic space **395**

16.1 The $\mathbf{L} \stackrel{?}{=} \mathbf{NL}$ problem 395
16.2 Alternation 399
16.3 Undirected reachability 401
16.4 Notes, references, and problems 405

PART V: BEYOND NP **409**

17 The polynomial hierarchy **411**

17.1 Optimization problems 411
17.2 The hierarchy 424
17.3 Notes, references, and problems 433

18 Computation that counts **439**

18.1 The permanent 439
18.2 The class $\oplus\mathbf{P}$ 447
18.3 Notes, references, and problems 452

19 Polynomial space **455**

19.1 Alternation and games 455
19.2 Games against nature and interactive protocols 469
19.3 More **PSPACE**-complete problems 480
19.4 Notes, references, and problems 487

20 A glimpse beyond **491**

20.1 Exponential time 491
20.2 Notes, references, and problems 499

Index **509**
Author index **519**

PART I

ALGORITHMS

Any book on algorithms ends with a chapter on complexity, so it is fitting to start this book by recounting some basic facts about algorithms. Our goal in these three chapters is to make a few simple but important points: Computational problems are not only things that have to be solved, they are also objects that can be worth studying. Problems and algorithms can be formalized and analyzed mathematically—for example as languages and Turing machines, respectively—and the precise formalism does not matter much. Polynomial-time computability is an important desirable property of computational problems, akin to the intuitive notion of practical solvability. Many different models of computation can simulate one another with a polynomial loss of efficiency—with the single exception of nondeterminism, which appears to require exponential time for its simulation. And there are problems that have no algorithms at all, however inefficient.

CHAPTER 1

PROBLEMS AND ALGORITHMS

An algorithm is a detailed step-by-step method for solving a problem. But what is a problem? We introduce in this chapter three important examples.

1.1 GRAPH REACHABILITY

A graph $G = (V, E)$ is a finite set V of nodes and a set E of edges, which are pairs of nodes (see Figure 1.1 for an example; all our graphs will be finite and directed). Many computational problems are concerned with graphs. The most basic problem on graphs is this: Given a graph G and two nodes $1, n \in V$, is there a path from 1 to n? We call this problem REACHABILITY[†]. For example, in Figure 1 there is indeed a path from node 1 to $n = 5$, namely $(1, 4, 3, 5)$. If instead we reverse the direction of edge $(4, 3)$, then no such path exists.

Like most interesting problems, REACHABILITY has an infinite set of possible *instances*. Each instance is a mathematical object (in our case, a graph and two of its nodes), of which we ask a question and expect an answer. The specific kind of question asked characterizes the problem. Notice that REACHABILITY asks a question that requires either a "yes" or a "no" answer. Such problems are called *decision problems*. In complexity theory we usually find it conveniently unifying and simplifying to consider only decision problems, instead of problems requiring all sorts of different answers. So, decision problems will play an important role in this book.

[†] In complexity theory, computational problems are not just things to solve, but also mathematical objects that are interesting in their own right. When problems are treated as mathematical objects, their names are shown in capital letters.

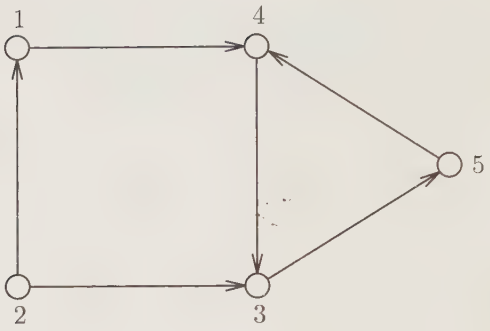


Figure 1-1. A graph.

We are interested in algorithms that solve our problems. In the next chapter we shall introduce the *Turing machine*, a formal model for expressing arbitrary algorithms. For the time being, let us describe our algorithms informally. For example, REACHABILITY can be solved by the so-called *search algorithm*. This algorithm works as follows: Throughout the algorithm we maintain a set of nodes, denoted S. Initially, $S = \{1\}$. Each node can be either *marked* or *unmarked*. That node i is marked means that i has been in S at some point in the past (or, it is presently in S). Initially, only 1 is marked. At each iteration of the algorithm, we choose a node $i \in S$ and remove it from S. We then process one by one all edges (i, j) out of i. If node j is unmarked, then we mark it, and add it to S. This process continues until S becomes empty. At this point, we answer "yes" if node n is marked, and "no" if it is not marked.

It should be clear that this familiar algorithm solves REACHABILITY. The proof would establish that a node is marked if and only if there is a path from 1 to it; both directions are easy inductions (see Problem 1.4.2). It is also clear, however, that there are important details that have been left out of our description of the algorithm. For example, how is the graph represented as an input to the algorithm? Since appropriate representations depend on the specific model of algorithms we use, this will have to wait until we have a specific model. The main point of that discussion (see Section 2.2) is that *the precise representation does not matter much*. You can assume in the meantime that the graph is given through its adjacency matrix (Figure 1.2), all entries of which can be accessed by the algorithm in a random access fashion‡.

There are also unclear spots in the algorithm itself: How is the element

‡ As a matter of fact, in Chapter 7 we shall see important applications of REACHABILITY to complexity theory in which the graph is given *implicitly;* that is, each entry of its adjacency matrix can be computed from the input data.

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Figure 1.2. Adjacency matrix.

$i \in S$ chosen among all elements of S? The choice here may affect significantly the style of the search. For example, if we always choose the node that has stayed in S the longest (in other words, if we implement S as a queue) then the resulting search is *breadth-first*, and a shortest path is found. If S is maintained as a stack (we choose the node added last), we have a kind of *depth-first* search. Other ways of maintaining S would result in completely different kinds of search. But the algorithm works correctly for all such choices.

Moreover, it works efficiently. To see that it does, notice that each entry of the adjacency matrix is visited only once, when the vertex corresponding to its row is chosen. Hence, we only spend about n^2 operations processing edges out of the chosen nodes (after all, there can be at most n^2 edges in a graph). Assuming that the other simple operations required (choosing an element of the set S, marking a vertex, and telling whether a vertex is marked) can each somehow be done in constant time, we conclude that the search algorithm determines whether two nodes in a graph with n nodes are connected in time at most proportional to n^2, or $\mathcal{O}(n^2)$.

The $\mathcal{O}$ notation we just used, and its relatives, are very useful in complexity theory, so we open here a brief parenthesis to define them formally.

Definition 1.1: We denote by $\mathbf{N}$ the set of all nonnegative integers. In complexity theory we concern ourselves with functions from $\mathbf{N}$ to $\mathbf{N}$, such as n^2, 2^n, and $n^3 - 2n + 5$. We shall use the letter n as the standard argument of such functions. Even when we denote our functions in a way that admits non-integer or negative values—such as $\sqrt{n}$, $\log n$, and $\sqrt{n} - 4\log^2 n$—we shall always think of the values as nonnegative integers. That is, any function denoted as $f(n)$, as in these examples, for us really means $\max\{\lceil f(n) \rceil, 0\}$.

So, let f and g be functions from $\mathbf{N}$ to $\mathbf{N}$. We write $f(n) = \mathcal{O}(g(n))$ (pronounced "$f(n)$ is big oh of $g(n)$," or "f is of the order of g") if there are positive integers c and n_0 such that, for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$. $f(n) = \mathcal{O}(g(n))$ means informally that f grows as g or slower. We write $f(n) = \Omega(g(n))$ if the opposite happens, that is, if $g(n) = \mathcal{O}(f(n))$. Finally, $f(n) = \Theta(g(n))$ means that $f = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n))$. The latter means that f and g have precisely the same *rate of growth*.

For example, it is easy to see that, if $p(n)$ is a polynomial of degree d, then

$p(n) = \Theta(n^d)$. That is, the rate of growth of a polynomial is captured by the polynomial's first non-zero term. The following is perhaps the most important and useful fact about growths of functions for complexity theory: If $c > 1$ is an integer and $p(n)$ any polynomial, then $p(n) = \mathcal{O}(c^n)$, but it is not the case that $p(n) = \Omega(c^n)$. That is, *any polynomial grows strictly slower than any exponential* (for the proof see Problem 1.4.9). One exponential lower, the same property implies that $\log n = \mathcal{O}(n)$ —in fact, that $\log^k n = \mathcal{O}(n)$ for any power k. □

Polynomial-Time Algorithms

Coming back to our $\mathcal{O}(n^2)$ algorithm for REACHABILITY, it should be no surprise that this simple problem can be solved satisfactorily by this simple algorithm—in fact, our estimate of $\mathcal{O}(n^2)$ is rather pessimistic, although in a way that we shall not consider important here; see Problem 1.4.3. It is important, however, to identify the source of our satisfaction: It is the rate of growth $\mathcal{O}(n^2)$. In the course of this book we shall regard such *polynomial* rates of growth as acceptable time requirements, as a sign that the problem has been solved satisfactorily. In contrast, *exponential* rates such as 2^n and, even worse, $n!$ will be a cause of concern. If they persist, and algorithm after algorithm we devise fails to solve the problem in polynomial time, we generally consider this as evidence that the problem in hand is perhaps *intractable*, not amenable to a practically efficient solution. Then the methods in this book come into play.

This dichotomy between polynomial and nonpolynomial time bounds, and the identification of polynomial algorithms with the intuitive notion of "practically feasible computation," is not uncontroversial. There are efficient computations that are not polynomial, and polynomial computations that are not efficient in practice†. For example, an n^{80} algorithm would probably be of limited practical value, and an algorithm with an exponential growth rate such as $2^{\frac{n}{100}}$ (or, more intriguing, a subexponential one such as $n^{\log n}$) may be far more useful.

There are, however, strong arguments in favor of the polynomial paradigm. First, it is a fact that any polynomial rate will be overcome eventually by any exponential one, and so the latter is to be preferred for all but a finite

† In fact, there is an important problem that provides examples for both kinds of exceptions: *Linear programming* (see the discussion and references in 9.5.34). A widely used classical algorithm for this basic problem, the *simplex method*, is known to be exponential in the worst case, but has consistently superb performance in practice; in fact, its expected performance is provably polynomial. In contrast, the first polynomial algorithm discovered for this problem, the *ellipsoid algorithm*, appears to be impractically slow. But the story of linear programming may in fact be a subtle argument for, not against, the methodology of complexity theory: It seems to indicate that problems that have practical algorithms are indeed polynomial-time solvable —although the polynomial-time algorithm and the empirically good one may not necessarily coincide.

set of instances of the problem — but of course this finite set may contain all instances that are likely to come up in practice, or that can exist within the confines of our universe... More to the point, experience with algorithms has shown that extreme rates of growth, such as n^{80} and $2^{\frac{n}{100}}$, rarely come up in practice. Polynomial algorithms typically have small exponents and reasonable multiplicative constants, and exponential algorithms are usually impractical indeed.

Another potential criticism of our point of view is that it only examines how the algorithm performs in the least favorable situations. The exponential worst-case performance of an algorithm may be due to a statistically insignificant fraction of the inputs, although the algorithm may perform satisfactorily *on the average*. Surely an analysis of the *expected*, as opposed to the worst-case, behavior of an algorithm would be more informative. Unfortunately, in practice we rarely know the *input distribution* of a problem — that is, the probability with which each possible instance is likely to occur as an input to our algorithm— and thus a truly informative average-case analysis is impossible. Besides, if we wish to solve just one particular instance, and our algorithm performs abysmally on it, knowing that we have stumbled upon a statistically insignificant exception is of little help or consolation[‡].

It should not come as a surprise that our choice of polynomial algorithms as the mathematical concept that is supposed to capture the informal notion of "practically efficient computation" is open to criticism from all sides. Any attempt, in any field of mathematics, to capture an intuitive, real-life notion (for example, that of a "smooth function" in real analysis) by a mathematical concept (such as C_∞) is bound to include certain undesirable specimens, while excluding others that arguably should be embraced. Ultimately, our argument for our choice must be this: *Adopting polynomial worst-case performance as our criterion of efficiency results in an elegant and useful theory that says something meaningful about practical computation, and would be impossible without this simplification.* Indeed, there is much mathematical convenience in polynomials: They form a stable class of functions, robust under addition, multiplication, and left and right substitution (see Problem 7.4.4); besides the logarithms of polynomial functions are all related by a constant—they are $\Theta(\log n)$ — and this will be handy in several occasions, such as in our discussion of space bounds.

Which brings us to the *space requirements* of our search algorithm for REACHABILITY. The algorithm basically needs to store the set S, and the "markings" of the nodes. There are at most n markings, and S cannot become larger than n; so, our algorithm uses $\mathcal{O}(n)$ space. Should we celebrate?

---

‡ In Chapter 11 and elsewhere, we do study algorithms from a probabilistic point of view; however, the source of randomness is *within the algorithm itself*, rather than some lottery of inputs. For an interesting complexity-theoretic treatment of average-case performance see Problem 12.3.10.

It turns out that, when space is concerned, we tend to be a little more stingy than with time—and sometimes are able to be a lot thriftier. As we shall see in Section 7.3, there is an algorithm for REACHABILITY, very different from depth-first search and breadth-first search (it could be called *middle-first search*), that uses substantially less space than n; in particular, it uses $\mathcal{O}(\log^2 n)$ space.

1.2 MAXIMUM FLOW

Our next example of a computational problem is a generalization of REACHABILITY. Accordingly, the input is a kind of generalized graph called a *network* (see Figure 1.3). A network $N = (V, E, s, t, c)$ is a graph (V, E) with two specified nodes s (the *source*) and t (the *sink*). We may assume that the source has no incoming edges, and that the sink has no outgoing edges. Also, for each edge (i, j) we are given a *capacity* $c(i, j)$, a positive integer. A *flow* in N is an assignment of a nonnegative integer value $f(i, j) \leq c(i, j)$ to each edge (i, j), such that for each node, other than s and t, the sum of the fs of the incoming edges is equal to the sum of the outgoing edges. The *value* of a flow f is the sum of the flows in the edges leaving s (or the sum arriving at t; these quantities are shown equal by adding the equations of flow conservation for all nodes). The MAX FLOW problem is this: Given a network N, find a flow of the largest possible value.

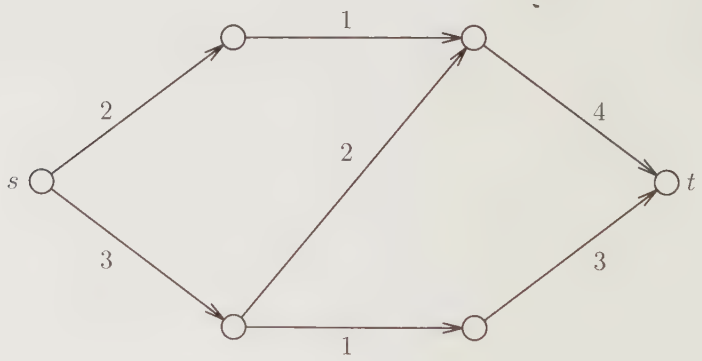


Figure 1-3. A network.

Obviously, MAX FLOW is not a decision problem, as it requires much more information in an answer than a mere "yes" or "no". It is an *optimization problem*, because it seeks the best among many possible solutions, according to a simple cost criterion. We can, however, transform any optimization problem into a roughly equivalent decision problem by supplying a *target value* for the

quantity to be optimized, and asking the question whether this value can be attained. For example, in the decision version of MAX FLOW, denoted MAX FLOW (D), we are given a network N and an integer K (the *goal*), and we are asked whether there is a flow of value K or more. This problem is roughly equivalent to MAX FLOW in the sense that, if we have a polynomial-time algorithm for MAX FLOW, then we certainly do for MAX FLOW (D), *and vice versa* (the latter, slightly less trivial, observation is attained by *binary search*, see Example 10.4).

A polynomial-time algorithm that solves MAX FLOW builds on a fundamental fact about networks, known as *the max-flow min-cut theorem* (and at the same time it provides a proof of the theorem, see Problem 1.4.1). Here is how the algorithm works: Suppose that we are given a flow f, and we simply want to know whether it is optimal or not. If there is a flow f' of value greater than that of f, then $\Delta f = f' - f$ is itself a flow with positive value. The only problem with Δf is that it may have a negative value on some edge (i, j). However, such a negative value can be thought of as a positive flow along an edge (j, i). Such "reverse" flow must be at most equal to $f(i, j)$. On the other hand, the positive components $\Delta f(i, j)$ must be at most $c(i, j) - f(i, j)$.

Another way of putting this is by saying that Δf is a flow in a derived network $N(f) = (V, E', s, t, c')$, where $E' = E - \{(i, j) : f(i, j) = c(i, j)\} \cup \{(i, j) : (j, i) \in E, f(j, i) > 0\}$, and $c'(i, j) = c(i, j) - f(i, j)$ for $(i, j) \in E$, and $c'(i, j) = f(j, i)$ for $(i, j) \in E' - E$†. For example, for the network in Figure 1.3 and the flow of Figure 1.4(a), $N(f)$ is shown in Figure 1.4(b). So, telling whether f is optimum is the same as deciding whether there is no positive flow in $N(f)$. But we know how to do this: Finding a positive flow in a network with positive capacities is just like determining whether there is a path from s to t: An instance of REACHABILITY!

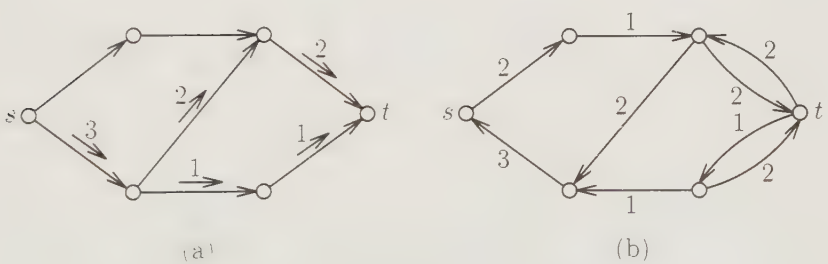


Figure 1-4. Construction of network $N(f)$.

† This assumes that there are no pairs of opposite arcs in N. Otherwise, a slightly more complex definition of $N(f)$ would do.

So, we can tell whether a flow is maximum. But this suggests the following algorithm for MAX FLOW: Start with the everywhere-zero flow in N. Repeatedly build $N(f)$, and attempt to find a path from s to t in $N(f)$. If there is a path, find the smallest capacity c' along its edges, and then add this amount of flow to the value of f on all edges appearing on this path; clearly the result is a larger flow. When there is no such path, we conclude that f is maximum and we are done.

How much time does this algorithm take to solve MAX FLOW? Each of the iterations of the algorithm (finding a path and augmenting the flow along it) takes $\mathcal{O}(n^2)$ time, as was established in the previous section. And there can be at most nC stages, where C is the maximum capacity of any edge in the network. The reason is this: Each iteration increases the flow by at least one (remember, the capacities are integers), and the maximum flow cannot be larger than nC (there are at most n edges going out of the source). So, there are at most nC iterations. In fact, as Figure 1.5 suggests, reality can be almost as bad: If our algorithm for REACHABILITY always returns a path involving edge (i, j) or edge (j, i), then $2C$ iterations are required. The total time for this algorithm is therefore $\mathcal{O}(n^3C)$.

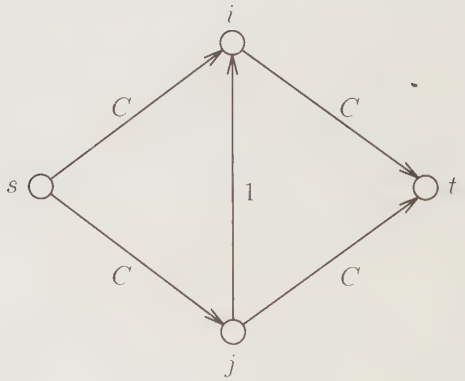


Figure 1-5. A bad example.

There is something annoying about this time bound, despite its perfectly polynomial appearance. The linear dependence of the running time of the algorithm on C is quite alarming. Numbers in an instance can get exceedingly high with little effort on the part of the provider of the instance (just by adding a few zeros). That is to say, numbers are usually represented *in binary* (or decimal, for that matter), and the length of such representations is logarithmic in the represented number. In this very real sense, the above bound (and

algorithm) *is not polynomial at all*, because its time requirements grow as an *exponential function of the length of the input.*

There is a simple and elegant way to overcome this difficulty: Suppose that we always augment the flow not along any path from s to t, but rather along the *shortest path*, that is, the one with the fewest edges. We can do this by using the *breadth-first version* of our search algorithm (notice how handily this stratagem defeats the "bad example" in Figure 1.5). An edge on a path from s to t in $N(f)$ that has the least capacity c' is called a *bottleneck*. Now, it is not hard to prove that, if we always choose the shortest path for augmenting the flow, each edge (i, j) of N, or its reverse (recall that each edge of $N(f)$ is either an edge of N or the reverse of one), *can be a bottleneck in at most* n *iterations* (see Problem 1.4.12). Since there are at most n^2 edges, and each iteration has at least one bottleneck, the number of iterations in the shortest-path version of the algorithm must be bounded by n^3. It follows immediately that this algorithm solves the MAX FLOW problem in time $\mathcal{O}(n^5)$.

Again, there are faster algorithms known for MAX FLOW, but they improve on the one explained above in ways that are not central to our interests here. These algorithms have time requirements that are bounded by a smaller polynomial than n^5. For example, it is known that a polynomial of the form n^3, or better, is possible; for *sparse* networks, with far fewer than n^2 edges, even faster algorithms are known, see the references in 1.4.13. Past experience in the field seems to suggest that, as a rule, once a polynomial algorithm for a problem has been developed, the time requirements undergo a series of improvements that bring the problem in the realm of realistic computation (MAX FLOW is a wonderful example, see the references in 1.4.13). The important step is to break the "barrier of exponentiality," to come up with the first polynomial time algorithm. We have just witnessed this breakthrough for MAX FLOW.

How about space? Even though each iteration of the algorithm can be performed in a small amount of space (recall the remark at the end of the previous section), we need a lot of extra space (about n^2) to store the current flow. We shall see much later in this book (Chapter 16) that no substantial improvement is likely.

Bipartite Matching

There is an interesting related problem that can be solved by similar techniques. Define a *bipartite graph* to be a triple $B = (U, V, E)$, where $U = \{u_1, \ldots, u_n\}$ is a set of nodes called *boys*, $V = \{v_1, \ldots, v_n\}$ is a set of *girls*, and $E \subseteq U \times V$ is a set of edges. Bipartite graphs are represented as in Figure 1.6(a) (notice that our bipartite graphs always have the same number of boys and girls). A *perfect matching*, or simply *matching*, in a bipartite graph is a set $M \subseteq E$ of n edges, such that, for any two edges $(u, v), (u', v') \in M$, $u \neq u'$ and $v \neq v'$; that is, no two edges in M are adjacent to the same boy, or the same girl (see

Figure 1.6(a) for an example). Intuitively, a matching is a way of assigning each boy to a different girl, so that, if u is assigned to v, then $(u, v) \in E$. Finally, MATCHING is the following computational problem: Given a bipartite graph, does it have a matching?

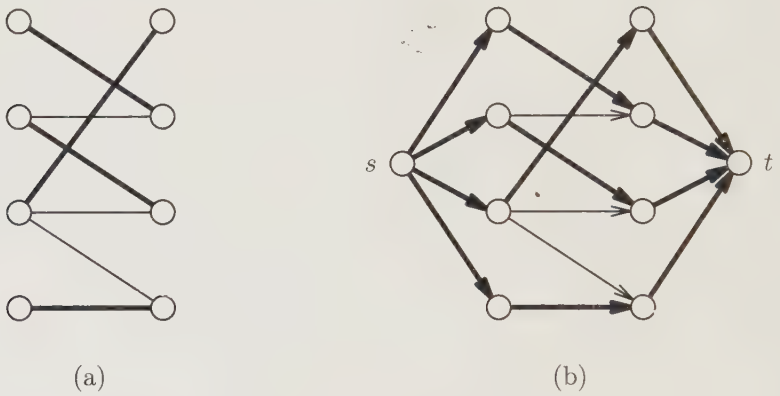


Figure 1-6. Bipartite graph (a) and associated network (b).

A central concept in algorithms is that of a *reduction.* A reduction is an algorithm that solves problem A by transforming any instance of A to an equivalent instance of a previously solved problem B. MATCHING provides an excellent illustration: Given any bipartite graph (U, V, E), we can construct a network with nodes $U \cup V \cup \{s, t\}$, where s is the source and t is the sink; with edges $\{(s, u) : u \in U\} \cup E \cup \{(v, t) : v \in V\}$; and with all capacities equal to one (see Figure 1.6(a) and (b) for an example). Then it is easy to see that *the original bipartite graph has a matching if and only if the resulting network has a flow of value n.* In other words, *we have reduced* MATCHING *to* MAX FLOW (D), a problem that we know how to solve in polynomial time. Hence, since the construction of the associated network can be done efficiently, MATCHING can be solved in polynomial time as well (in fact, for networks constructed as above for solving matching problems, it is trivial that the number of iterations is at most n, and hence the time required by the naive augmentation algorithm is $\mathcal{O}(n^3)$; faster algorithms are possible—see the references in 1.4.13).

Incidentally, reductions are very important for studying algorithms and complexity. In fact, a central tool of complexity theory is a perverse use of reduction, in which a problem is reduced not to an already-solved one, but *to a problem that we wish to show is difficult;* see Chapter 8 and beyond.

1.3 THE TRAVELING SALESMAN PROBLEM

The three problems discussed so far in this chapter (REACHABILITY, MAX FLOW, and MATCHING) represent some of the happiest moments in the theory of algorithms. We now come to another classical problem which has been that theory's most outstanding and persistent failure. It has also provided much motivation for the development of complexity theory. The problem is simply this: We are given n cities $1, \ldots, n$, and a nonnegative integer distance d_{ij} between any two cities i and j (assume that the distances are symmetric, that is, $d_{ij} = d_{ji}$ for all i and j). We are asked to find the *shortest tour* of the cities—that is, the permutation π such that $\sum_{i=1}^{n} d_{\pi(i),\pi(i+1)}$ (where by $\pi(n+1)$ we mean $\pi(1)$) is as small as possible. We call this problem TSP. Its decision version, TSP (D), is defined analogously with MAX FLOW (D). That is, an integer bound B (the traveling salesman's "budget") is provided along with the matrix d_{ij}, and the question is whether there is a tour of length at most B.

Obviously, we can solve this problem by enumerating all possible solutions, computing the cost of each, and picking the best. This would take time proportional to $n!$ (there are $\frac{1}{2}(n-1)!$ tours to be considered), which is not a polynomial bound at all. Let us note in passing that this naive algorithm fares reasonably well in terms of space: It requires space proportional to n, since we need remember only the permutation currently examined, and the best tour seen so far.

There are no polynomial-time algorithms known for TSP. We can improve on the $n!$ time bound a little by a dynamic programming method (see Problem 1.4.15; interestingly, the space needed becomes exponential!). There are heuristic algorithms that perform well, and return tours that are not too far from the optimum when supplied with "typical" instances. But, if we insist on algorithms that are guaranteed to compute the optimum, the current state of the art has only exponential answers to offer. This failure is not a result of lack of interest, since the TSP is one of the most intensively studied problems in mathematics.

It is tempting to suppose that a fundamental obstruction is present, a dramatic dividing line between the TSP and the other problems we have seen. We might suppose that *there can be no polynomial-time algorithm for* TSP. This assertion is one of the many ways of stating the $\mathbf{P} \neq \mathbf{NP}$ conjecture, one of the most important problems in computer science, and the central issue in this book.

1.4 NOTES, REFERENCES, AND PROBLEMS

1.4.1 For a much more thorough introduction to the subject of algorithms see the following books, among many others:

- D. E. Knuth *The Art of Computer Programming, volumes 1–3*, Addison-Wesley, Reading, Massachusetts, 1968–;
- A. V. Aho, J. E. Hopcroft, and J. D. Ullman *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachusetts, 1974;
- C. H. Papadimitriou and K. Steiglitz *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, New Jersey, 1982;
- T. H. Cormen, C. E. Leiserson, and R. L. Rivest *Introduction to Algorithms*, MIT Press, Cambridge, Massachusetts, 1990.

1.4.2 Problem: (a) Show by induction on i that, if v is the ith node added by the search algorithm of Section 1.1 to the set S, then there is a path from node 1 to v.

(b) Show by induction on ℓ that if node v is reachable from node 1 via a path with ℓ edges, then the search algorithm will add v to set S.

1.4.3 Problem: Show that in the search algorithm each edge of G is processed at most once. Conclude that the algorithm can be implemented to run in $\mathcal{O}(|E|)$ steps.

1.4.4 Problem: (a) A directed graph is *acyclic* if it has no cycles. Show that any acyclic graph has a source (a node with no incoming edges).

(b) Show that a graph with n nodes is acyclic if and only if its nodes can be numbered 1 to n so that all edges go from lower to higher numbers (use the property in (a) above repeatedly).

(c) Describe a polynomial-time algorithm that decides whether a graph is acyclic. (Implement the idea in (b) above. With care, your algorithm should not spend more than a constant amount of time per edge.)

1.4.5 Problem: (a) Show that a graph is bipartite (that is, its nodes can be partitioned into two sets, not necessarily of equal cardinality, with edges going only from one to the other) if and only if it has no odd-length cycles.

(b) Describe a polynomial algorithm for testing whether a graph is bipartite.

1.4.6 According to Lex Schrijver, an early record (if not the earliest) of a conscious analysis of the time requirements of an algorithm is

- G. Lamé "Note sur la limite du nombre des divisions dans la récherche du plus grand commun diviseur entre deux nombres entiers (A note on the bound on the number of divisions in the search for the greatest common divisor of two whole numbers)," *Com. Rend. des Séances de l'Acad. des Sciences, Paris, 19*, pp. 867–870, 1884.

This result, stating that Euclid's algorithm is polynomial *in the number of bits* of the integers, is the subject of Lemma 11.7 and Problem 11.5.8.

1.4.7 The germinal idea that polynomial time is a desirable property of computational problems, akin to practical solvability, can be found in the work of several researchers in logic, the theory of computation, and optimization during the 1950s and 60s (not to mention the previous reference):

- J. von Neumann "A certain zero-sum two-person game equivalent to the assignment problem," in *Contributions to the Theory of Games, 2*, pp. 5–12, edited by H. W. Kuhn and A. W. Tucker, Princeton Univ. Press, 1953;
- M. O. Rabin "Degree of difficulty of computing a function and a partial ordering of recursive sets," Tech. Rep. No 2, Hebrew Univ., 1960;
- M. O. Rabin "Mathematical theory of automata," *Proc. 19th Symp. Applied Math.*, pp. 153–175, 1966;
- A. Cobham "The intrinsic computational difficulty of functions," *Proceedings of the 1964 Congress on Logic, Mathematics and the Methodology of Science*, pp. 24-30, North Holland, New York, 1964;
- J. Edmonds "Paths, trees, and flowers," *Can J. Math, 17*, 3, pp. 449–467, 1965;
- J. Edmonds "Systems of distinct representatives and linear algebra," and "Optimum branchings," *J. Res. National Bureau of Standards, Part B, 17B*, 4, pp. 241–245 and 233–240, 1966–1967.

Jack Edmonds used the term "good algorithm" for polynomial time; in fact, the last two papers contain an informal discussion of **NP**, and the conjecture that the traveling salesman problem is not in **P**. For an earlier statement of the opposite conjecture by Kurt Gödel see 8.4.9. See also

- B. A. Trakhtenbrot "A survey of Russian approaches to *perebor* (brute-force search) algorithms," *Annals of the History of Computing, 6* pp. 384–400, 1984.

for an account of similar insights by researchers in the Soviet Union.

1.4.8 The version of the $\mathcal{O}$-notation we use in this book was proposed in

- D. E. Knuth "Big omicron and big omega and big theta," *ACM SIGACT News, 8*, 2, pp. 18–24, 1976.

1.4.9 Problem: Show that for any polynomial $p(n)$ and any constant c there is an integer n_0 such that, for all $n \geq n_0$, $2^{cn} > p(n)$. Calculate this n_0 when (a) $p(n) = n^2$ and $c = 1$; (b) when $p(n) = 100n^{100}$ and $c = \frac{1}{100}$.

1.4.10 Problem: Let $f(g)$ and $g(n)$ be any two of the following functions. Determine whether (i) $f(n) = \mathcal{O}(g(n))$; (ii) $f(n) = \Omega(g(n))$; or (iii) $f(n) = \Theta(g(n))$:

(a) n^2; (b) n^3; (c) $n^2 \log n$

(d) 2^n; (e) n^n; (f) $n^{\log n}$

(g) 2^{2^n}; (h) $2^{2^{n+1}}$; (j) n^2 is n is odd, 2^n otherwise.

1.4.11 The max-flow min-cut theorem: In any network the value of the maximum flow equals the capacity of the minimum cut.

This important result was established independently by

- L. R. Ford, D. R. Fulkerson *Flows in Networks*, Princeton Univ. Press, Princeton, N.J., 1962, and
- P. Elias, A. Feinstein, C. E. Shanon "Note on maximum flow through a network," *IRE Trans. Inform. Theory, IT-2*, pp. 117–119, 1956.

Here by a *cut* in a network $N = (V, E, s, t, c)$ we mean a set S of nodes such that $s \in S$ and $t \notin S$. The *capacity of the cut* S is the sum of the capacities of all edges going out of S.

Give a proof of the max-flow min-cut theorem based on the augmentation algorithm of Section 1.2. (The value of any flow is always at least as large as the capacity of any cut. And, when the algorithm terminates, it suggests a cut which is at least as large as the flow.)

1.4.12 Problem: Show that, if in the augmentation algorithm of Section 1.2 we always augment by a shortest path, then (a) the distance of a node from s cannot decrease from an $N(f)$ to the next. Furthermore, (b) if edge (i, j) is the bottleneck at one stage (the edge along the augmenting path with the smallest capacity), then the distance of i from s must increase before it becomes a bottleneck in the opposite direction. (c) Conclude that there are at most $\mathcal{O}(|E||V|)$ augmentations.

1.4.13 Over the past twenty years, a long sequence of more and more efficient algorithms for MAX FLOW has been proposed in the literature. The augmentation algorithm in the book by Ford and Fulkerson cited above is, of course, exponential. The idea of shortest-path augmentation above, resulting in an $\mathcal{O}(|E|^2|V|)$ algorithm, was proposed in

- J. Edmonds and R. M. Karp "Theoretical improvements in algorithmic efficiency for network flow problems," *J.ACM, 19*, 2, pp. 248–264, 1972.

An $\mathcal{O}(|E||V|^2)$ algorithm results if we further explore this idea, and simultaneously augment along *many* shortest paths in the same phase:

- E. A. Dinic "Algorithm for solution of a problem of maximal flow in a network with [efficiency analysis]," *Soviet Math. Doklady*, 11, pp. 1277–1280, 1970

The same idea gives an $\mathcal{O}(n^{2.5})$ algorithm for matching:

- J. E. Hopcroft, R. M. Karp "An $n^{\frac{5}{2}}$ algorithm for maximum matchings in bipartite graphs," *SIAM J. Comp., 2*, 4, pp. 225–231, 1973.

Still faster algorithms for MAX FLOW allow intermediate "flows" that do not satisfy the conservation constraints at each node; the time complexity is $\mathcal{O}(|V|^3)$. For a nice exposition see:

- V. M. Malhotra, M. P. Kumar, and S. N. Maheshwari "An $\mathcal{O}(|V|^3)$ algorithm for finding maximal flows in networks," *Information Processing Letters 7*, 6, pp. 277-278, 1978.

But even faster algorithms are known. For sparse networks, the asymptotically fastest known algorithm is based on a novel elaboration on the intermediate "flow" idea above, and takes time $\mathcal{O}(|E||V| \log \frac{|V|^2}{|E|})$:

- A. V. Goldberg and R. E. Tarjan "A new approach to the maximum flow problem," *Proc. of the 18th Annual ACM Symposium on the Theory of Computing*, pp. 136–146, 1986.

For a survey of algorithms for MAX FLOW and its generalizations see

- A. V. Goldberg, É. Tardos, and R. E. Tarjan "Network flow algorithms," Technical report STAN-CS-89-1252, Computer Science Department, Stanford University, 1989.

1.4.14 There is a generalization of matching to general, non-bipartite graphs, as follows: Call a set of node-disjoint edges in a graph a *matching*. A *complete matching* is then one that covers all nodes (obviously, only graphs with an even number of nodes can have one). Notice that, when restricted to a graph with no odd cycles, this problem degenerates to our matching problem. But odd cycles present some serious problems in generalizing our max-flow technique. There is a much more sophisticated $\mathcal{O}(n^5)$ algorithm for the general matching problem due to Jack Edmonds (see his 1965 paper cited in Note 1.4.7). As usual, its time requirements have been improved tremendously over the years.

1.4.15 Problem: Suppose that we are given an instance of the TSP with n cities and distances d_{ij}. For each subset S of the cities *excluding city* 1, and for each $j \in S$, define $c[S, j]$ to be the shortest path that starts from city 1, visits all cities in S and ends up in city j.

(a) Give an algorithm that calculates $c[S, j]$ *by dynamic programming*, that is, progressing from smaller to larger sets S.

(b) Show that this algorithm solves the TSP in time $\mathcal{O}(n^2 2^n)$. What are the space requirements of the algorithm?

(c) Suppose we wish to find the shortest (in the sense of sum of weights) path from 1 to n, not necessarily visiting all cities. How would you modify the above algorithm? (Is the reference to S necessary now, or can it be replaced by simply $|S|$?) Show that this problem can be solved in polynomial time.

CHAPTER 2

TURING MACHINES

Despite its weak and clumsy appearance, the Turing machine can simulate arbitrary algorithms with inconsequential loss of efficiency. It will be our formal model for algorithms in this book.

2.1 TURING MACHINE BASICS

It is amazing how little we need to have everything! Viewed as a programming language, the Turing machine has a single data structure, and rather primitive one at that: A string of symbols. The available operations allow the program to move a cursor left and right on the string, to write on the current position, and to branch depending on the value of the current symbol. All in all, it is an extremely weak and primitive language. And yet, we shall argue in this chapter, it is capable of expressing any algorithm, of simulating any programming language.

Definition 2.1: Formally, a Turing machine is a quadruple $M = (K, \Sigma, \delta, s)$. Here K is a finite set of *states* (these are the instructions alluded to above); $s \in K$ is the *initial state*. Σ is a finite set of *symbols* (we say Σ is the *alphabet* of M). We assume that K and Σ are disjoint sets. Σ always contains the special symbols $\sqcup$ and $\triangleright$: The *blank* and the *first symbol*. Finally, δ is a *transition function*, which maps $K \times \Sigma$ to $(K \cup \{h, \text{“yes”}, \text{“no”}\}) \times \Sigma \times \{\leftarrow, \rightarrow, -\}$. We assume that h (the *halting state*), “yes” (the *accepting state*), “no” (the *rejecting state*), and the *cursor directions* $\leftarrow$ for “left,” $\rightarrow$ for “right,” and $-$ for “stay,” are not in $K \cup \Sigma$. □

Function δ is the “program” of the machine. It specifies, for each combination of current state $q \in K$ and current symbol $\sigma \in \Sigma$, a triple $\delta(q, \sigma) =$

(p, ρ, D). p is the next state, ρ is the symbol to be overwritten on σ, and $D \in \{\leftarrow, \rightarrow, -\}$ is the direction in which the cursor will move. For $\triangleright$ we require that, if for states q and p, $\delta(q, \triangleright) = (p, \rho, D)$, then $\rho = \triangleright$ and $D = \rightarrow$. That is, $\triangleright$ always directs the cursor to the right, and is never erased.

How is the program to start? Initially the state is s. The string is initialized to a $\triangleright$, followed by a finitely long string $x \in (\Sigma - \{\sqcup\})^*$. We say that x is the *input* of the Turing machine. The cursor is pointing to the first symbol, always a $\triangleright$.

From this initial configuration the machine takes a step according to δ, changing its state, printing a symbol, and moving the cursor; then it takes another step, and another. Note that, by our requirement on $\delta(p, \triangleright)$, the string will always start with a $\triangleright$, and thus the cursor will never "fall off" the left end of the string.

Although the cursor will never fall off the left end, it will often wander off the right end of the string. In this case we think that the cursor scans a $\sqcup$, which of course may be overwritten immediately. This is how the string becomes longer—a necessary feature, if we wish our machines to perform general computation. The string never becomes shorter.

Since δ is a completely specified function, and the cursor never falls off the left end, there is only one reason why the machine cannot continue: One of the three halting states h, "yes", and "no" has been reached. If this happens, we say that the machine has *halted*. Furthermore, if state "yes" has been reached, we say the machine *accepts* its input; if "no" has been reached, then it *rejects* its input. If a machine halts on input x, we can define the *output* of the machine M on x, denoted $M(x)$. If M accepts or rejects x, then $M(x)$ = "yes" or "no", respectively. Otherwise, if h was reached, then the output is the string of M at the time of halting. Since the computation has gone on for finitely many steps, the string consists of a $\triangleright$, followed by a finite string y, whose last symbol is not a $\sqcup$, possibly followed by a string of $\sqcup$s (y could be empty). *We consider string y to be the output of the computation*, and write $M(x) = y$. Naturally, it is possible that M will never halt on input x. If this is the case we write $M(x) = \nearrow$.

Example 2.1: Consider the Turing machine $M = (K, \Sigma, \delta, s)$, where $K = \{s, q, q_0, q_1\}$, $\Sigma = \{0, 1, \sqcup, \triangleright\}$, and δ is as in Figure 2.1. If we start this machine with input 010, the computation also shown in Figure 2.1 takes place. The reader should examine closely the operation of this machine on this input, and speculate about its behavior on any other input. It is not hard to see that M simply *inserts a $\sqcup$ between $\triangleright$ and its input* (see Problem 2.8.2). It always halts.

It is, however, true that, if the machine ever executes the "instruction" $\delta(q, \sqcup) = (q, \sqcup, -)$ (seventh line in the table on the left of Figure 2.1), it will keep executing the same instruction over and over again, and it will never halt. The point is that this transition will never be executed if the machine starts

in accordance with our convention, with a string of the form $\triangleright x$, with no $\sqcup$s within x, and with the cursor at $\triangleright$. We could have defined $\delta(q, \sqcup)$ to be any other triple, and the machine would be precisely equivalent. In our descriptions of Turing machines we shall feel free to omit such irrelevant transitions. □

$p \in K$,	$\sigma \in \Sigma$	$\delta(p, \sigma)$
s,	0	$(s, 0, \rightarrow)$
s,	1	$(s, 1, \rightarrow)$
s,	$\sqcup$	$(q, \sqcup, \leftarrow)$
s,	$\triangleright$	$(s, \triangleright, \rightarrow)$
q,	0	$(q_0, \sqcup, \rightarrow)$
q,	1	$(q_1, \sqcup, \rightarrow)$
q,	$\sqcup$	$(q, \sqcup, -)$
q,	$\triangleright$	$(h, \triangleright, \rightarrow)$
q_0,	0	$(s, 0, \leftarrow)$
q_0,	1	$(s, 0, \leftarrow)$
q_0,	$\sqcup$	$(s, 0, \leftarrow)$
q_0,	$\triangleright$	$(h, \triangleright, \rightarrow)$
q_1,	0	$(s, 1, \leftarrow)$
q_1,	1	$(s, 1, \leftarrow)$
q_1,	$\sqcup$	$(s, 1, \leftarrow)$
q_1,	$\triangleright$	$(h, \triangleright, \rightarrow)$

0.	s,	$\underline{\triangleright}010$
1.	s,	$\triangleright\underline{0}10$
2.	s,	$\triangleright 0\underline{1}0$
3.	s,	$\triangleright 01\underline{0}$
4.	s,	$\triangleright 010\underline{\sqcup}$
5.	q,	$\triangleright 01\underline{0}\sqcup$
6.	q_0,	$\triangleright 01\,\sqcup\,\sqcup$
7.	s,	$\triangleright 01\underline{\sqcup}0$
8.	q,	$\triangleright 0\underline{1}\,\sqcup\,0$
9.	q_1,	$\triangleright 0\,\sqcup\,\underline{\sqcup}0$
10.	s,	$\triangleright 0\underline{\sqcup}10$
11.	q,	$\triangleright\underline{0}\,\sqcup\,10$
12.	q_0,	$\triangleright\,\sqcup\,\underline{\sqcup}10$
13.	s,	$\triangleright\underline{\sqcup}010$
14.	q,	$\underline{\triangleright}\,\sqcup\,010$
15.	h,	$\triangleright\underline{\sqcup}010$

Figure 2.1. Turing machine and computation.

We can define the operation of a Turing machine formally using the notion of a *configuration*. Intuitively, a configuration contains a complete description of the current state of the computation, all information displayed in each line of the right-hand part of Figure 2.1. Formally, a configuration of M is a triple (q, w, u), where $q \in K$ is a state, and w, u are strings in Σ^*. w is the string to the left of the cursor, including the symbol scanned by the cursor, and u is the string to the right of the cursor, possibly empty. q is the current state. For example, the machine in Figure 2.1 starts at the configuration $(s, \triangleright, 010)$. It subsequently enters configurations $(s, \triangleright 0, 10)$, $(s, \triangleright 01, 0)$, $(s, \triangleright 010, \epsilon)$, $(s, \triangleright 010\sqcup, \epsilon)$, $(q, \triangleright 010, \sqcup)$, $(q_0, \triangleright 01\,\sqcup\,\sqcup, \epsilon)$, etc. ($\epsilon$ denotes the empty string).

Definition 2.2: Let us fix a Turing machine M. We say that configuration (q, w, u) *yields* configuration (q', w', u') *in one step*, denoted $(q, w, u) \xrightarrow{M} (q', w', u')$, intuitively if a step of the machine from configuration (q, w, u) results in configuration (q', w', u'). Formally, it means that the following holds. First, let σ be the last symbol of w, and suppose that $\delta(q, \sigma) = (p, \rho, D)$. Then

we must have that $q' = p$. We have three cases. If $D = \rightarrow$, then w' is w with its last symbol (which was a σ) replaced by ρ, and the first symbol of u appended to it ($\sqcup$ if u is the empty string); u' is u with the first symbol removed (or, if u was the empty string, u' remains empty). If $D = \leftarrow$, then w' is w with σ omitted from its end, and u' is u with ρ attached in the beginning. Finally, if $D = -$, then w' is w with the ending σ replaced by ρ, and $u' = u$.

Once we defined the relationship of "yields in one step" among configurations, we can define "yields" to be its transitive closure. That is, we say that configuration (q, w, u) *yields* configuration (q', w', u') *in* k *steps*, denoted $(q, w, u) \overset{M}{\rightarrow}^{k} (w', q', u')$, where $k \geq 0$ is an integer, if there are configurations (q_i, w_i, u_i), $i = 1, \ldots, k+1$, such that $(q_i, w_i, u_i) \overset{M}{\rightarrow} (q_{i+1}, w_{i+1}, u_{i+1})$ for $i = 1, \ldots, k$, $(q_1, w_1, u_1) = (q, w, u)$, and $(q_{k+1}, w_{k+1}, u_{k+1}) = (q', w', u')$. Finally, we say that configuration (q, w, u) *yields* configuration (q', w', u'), denoted $(q, w, u) \overset{M}{\rightarrow}^{*} (q', w', u')$, if there is a $k \geq 0$ such that $(q, w, u) \overset{M}{\rightarrow}^{k} (q', w', u')$. □

Example 2.1 (Continued): In Figure 2.1 we have $(s, \triangleright, 010) \overset{M}{\rightarrow} (s, \triangleright 0, 10)$, $(s, \triangleright, 010) \overset{M}{\rightarrow}^{15} (h, \triangleright \sqcup, 010)$, and thus $(s, \triangleright, 010) \overset{M}{\rightarrow}^{*} (h, \triangleright \sqcup, 010)$. □

$p \in K,$	$\sigma \in \Sigma$	$\delta(p, \sigma)$
$s,$	0	$(s, 0, \rightarrow)$
$s,$	1	$(s, 1, \rightarrow)$
$s,$	$\sqcup$	$(q, \sqcup, \leftarrow)$
$s,$	$\triangleright$	$(s, \triangleright, \rightarrow)$
$q,$	0	$(h, 1, -)$
$q,$	1	$(q, 0, \leftarrow)$
$q,$	$\triangleright$	$(h, \triangleright, \rightarrow)$

Figure 2.2. Turing machine for binary successor.

Example 2.2: Consider the Turing machine in Figure 2.2. If its input is an integer n in binary (possibly with leading 0s), the machine computes the binary representation of $n + 1$. It first moves right with state s to find the least significant bit of n, and then goes to the left examining bits, in order of increasing significance. As long as it sees a 1, it turns it into a 0, and continues to the left, to take care of the carry. When it sees a 0, it turns it into a 1 and halts.

On input 11011, the computation is this: $(s, \triangleright, 11011) \overset{M}{\rightarrow} (s, \triangleright 1, 1011) \overset{M}{\rightarrow} (s, \triangleright 11, 011) \overset{M}{\rightarrow} (s, \triangleright 110, 11) \overset{M}{\rightarrow} (s, \triangleright, 1101, 1) \overset{M}{\rightarrow} (s, \triangleright 11011, \epsilon) \overset{M}{\rightarrow} (s, \triangleright 11011 \sqcup, \epsilon) \overset{M}{\rightarrow} (q, \triangleright 11011, \sqcup) \overset{M}{\rightarrow} (q, \triangleright 1101, 0 \sqcup) \overset{M}{\rightarrow} (q, \triangleright 110, 00 \sqcup) \overset{M}{\rightarrow} (h, \triangleright 111, 00 \sqcup)$. On in-

put 111 the machine will do this: $(s, \triangleright, 111) \overset{M}{\to} (s, \triangleright 1, 11) \overset{M}{\to} (s, \triangleright 11, 1) \overset{M}{\to} (s, \triangleright 111, \epsilon) \overset{M}{\to} (s, \triangleright 111\sqcup, \epsilon) \overset{M}{\to} (q, \triangleright 111, \sqcup) \overset{M}{\to} (q, \triangleright 11, 0\sqcup) \overset{M}{\to} (q, \triangleright 1, 00\sqcup) \overset{M}{\to} (q, \triangleright, 000\sqcup) \overset{M}{\to} (h, 0, 00\sqcup)$.

There is a "bug" in this machine: On input 1^k (that is, on the binary representation of the integer $2^k - 1$) it will "overflow" and answer zero. (What will it do on input ϵ?) One way around this problem would be to first employ the machine in Figure 2.1 as a "subroutine" that shifts the input one position to the right and then add a leading 0 to the resulting input. Then the *overall machine* will correctly compute $n + 1$. □

Example 2.3: The two machines we have shown so far compute some simple functions from strings to strings. Accordingly, they always end up at state h, the halting state that signals the output is ready. We shall next see a machine which computes no output; it only signals its approval or disapproval of its input by halting at state "yes" or "no".

The purpose of the machine shown in Figure 2.3 is to tell whether its input is a *palindrome*, that is, whether it reads the same backwards, just like the string $\nu\iota\psi o\nu\alpha\nu o\mu\eta\mu\alpha\tau\alpha\mu\eta\mu o\nu\alpha\nu o\psi\iota\nu$. If M's input is a palindrome, M accepts (ends up at state "yes"). If its input is not a palindrome, M halts on "no" and rejects. For example, $M(0000) =$ "yes", and $M(011) =$ "no".

$p \in K$,	$\sigma \in \Sigma$	$\delta(p, \sigma)$
s	0	$(q_0, \triangleright, \rightarrow)$
s	1	$(q_1, \triangleright, \rightarrow)$
s	$\triangleright$	$(s, \triangleright, \rightarrow)$
s	$\sqcup$	("yes", $\sqcup$, $-$)
q_0	0	$(q_0, 0, \rightarrow)$
q_0	1	$(q_0, 1, \rightarrow)$
q_0	$\sqcup$	$(q_0', \sqcup, \leftarrow)$
q_1	0	$(q_1, 0, \rightarrow)$
q_1	1	$(q_1, 1, \rightarrow)$
q_1	$\sqcup$	$(q_1', \sqcup, \leftarrow)$

$p \in K$,	$\sigma \in \Sigma$	$\delta(p, \sigma)$
q_0'	0	$(q, \sqcup, \leftarrow)$
q_0'	1	("no", 1, $-$)
q_0'	$\triangleright$	("yes", $\sqcup$, $\rightarrow$)
q_1'	0	("no", 1, $-$)
q_1'	1	$(q, \sqcup, \leftarrow)$
q_1'	$\triangleright$	("yes", $\triangleright$, $\rightarrow$)
q	0	$(q, 0, \leftarrow)$
q	1	$(q, 1, \leftarrow)$
q	$\triangleright$	$(s, \triangleright, \rightarrow)$

Figure 2.3. Turing machine for palindromes.

The machine works as follows: In state s, it searches its string for the first symbol of the input. When it finds it, it makes it into a $\triangleright$ (thus effectively moving the left end of the string inward) and remembers it in its state. By this we mean that M enters state q_0 if the first symbol is a 0, and state q_1 if it is a 1 (this important capability of Turing machines to remember finite information in their state will be used over and over). M then moves to the right until the

first $\sqcup$ is met, and then once to the left to scan the last symbol of the input (now M is in state q_0' or q_1', still remembering the first symbol). If this last symbol agrees with the one remembered, it is replaced with a $\sqcup$ (so that the string implodes on the right as well). Then the rightmost $\triangleright$ is found using a new state q, and then the process is repeated. Notice that, as the two boundaries of the string (a $\triangleright$ on the left, a $\sqcup$ on the right) have "marched inwards," the string left is precisely the string that remains to be shown a palindrome. If at some point the last symbol is different from the first symbol as remembered by the machine, then the string is not a palindrome, and we reach state "no". If we end up with the empty string (or we fail to find the last symbol, which means that the string was a single symbol) we express our approval by "yes".

On input 0010, the following configurations, among others, will be yielded: $(s, \triangleright, 0010) \overset{M}{\to}^5 (q_0, \triangleright\triangleright 010\sqcup, \epsilon) \overset{M}{\to} (q_0', \triangleright\triangleright 010, \sqcup) \overset{M}{\to} (q, \triangleright\triangleright 01, \sqcup\sqcup) \overset{M}{\to}^2 (q, \triangleright\triangleright, 01\sqcup\sqcup) \overset{M}{\to} (s, \triangleright\triangleright 0, 1\sqcup\sqcup) \overset{M}{\to} (q_0, \triangleright\triangleright\triangleright, 1, \sqcup\sqcup) \overset{M}{\to}^2 (q_0, \triangleright\triangleright\triangleright 1\sqcup, \sqcup) \overset{M}{\to} (q_0', \triangleright\triangleright\triangleright 1, \sqcup\sqcup) \overset{M}{\to}$ ("no", $\triangleright\triangleright\triangleright 1, \sqcup\sqcup$).

On input 101, the computation is as follows: $(s, \triangleright, 101) \overset{M}{\to} (s, \triangleright 1, 01) \overset{M}{\to} (q_1, \triangleright\triangleright 0, 1) \overset{M}{\to}^3 (q_1', \triangleright\triangleright 01, \sqcup) \overset{M}{\to} (q, \triangleright\triangleright 0, \sqcup\sqcup) \overset{M}{\to} (q\ \triangleright\triangleright, 0\sqcup\sqcup) \overset{M}{\to} (s, \triangleright\triangleright 0, \sqcup\sqcup) \overset{M}{\to} (q_0, \triangleright\triangleright\triangleright, \sqcup\sqcup) \overset{M}{\to} (q_0, \triangleright\triangleright\triangleright\sqcup, \sqcup) \overset{M}{\to} (q_0', \triangleright\triangleright\triangleright, \sqcup\sqcup) \overset{M}{\to}$ ("yes", $\triangleright\triangleright\triangleright\sqcup, \sqcup$).

On input ϵ (the shortest palindrome in the world) here is the computation: $(s, \triangleright, \epsilon) \overset{M}{\to} (s, \triangleright\sqcup, \epsilon) \overset{M}{\to}$ ("yes", $\triangleright\sqcup, \epsilon$). □

2.2 TURING MACHINES AS ALGORITHMS

Turing machines seem ideal for solving certain specialized kinds of problems on strings, namely *computing string functions* and *accepting* and *deciding languages*. Let us define these tasks precisely.

Definition 2.3: Let $L \subset (\Sigma - \{\sqcup\})^*$ be a language, that is, a set of strings of symbols. Let M be a Turing machine such that, for any string $x \in (\Sigma - \{\sqcup\})^*$, if $x \in L$, then $M(x)$ = "yes" (that is, M on input x halts at the "yes" state), and, if $x \notin L$, then $M(x)$ = "no". Then we say that M *decides* L. If L is decided by some Turing machine M, then L is called a *recursive language*. For example, palindromes over $\{0,1\}^*$ constitute a recursive language decided by machine M in Figure 2.3.

We say that M simply *accepts* L whenever, for any string $x \in (\Sigma - \{\sqcup\})^*$, if $x \in L$, then $M(x)$ = "yes"; however, if $x \notin L$, then $M(x) = \nearrow$. If L is accepted by some Turing machine M, then L is called *recursively enumerable*. □

Let us immediately notice the intriguing *asymmetry* in the definition of acceptance by Turing machines. We obtain a useful outcome (halting on "yes") only in the case $x \in L$. If $x \notin L$, the machine computes away forever. Practically speaking, this is not a useful answer, since we will never know when we have

waited enough to be sure that the machine will not halt. As a result, acceptance is not a true algorithmic concept, *only a useful way of categorizing problems.* We shall see this pattern once more when, later in this chapter, we shall introduce *nondeterminism.*

We have not yet seen Turing machines that accept languages; but let us observe the following:

Proposition 2.1: If L is recursive, then it is recursively enumerable.

Proof: Suppose that there is a Turing machine M that decides L. We shall construct from M a Turing machine M' that *accepts* L, as follows: M' behaves exactly like M. Except that, whenever M is about to halt and enter state "no", M' moves to the right forever, and never halts. □

Definition 2.3 (Continued): We shall not only deal with the decision and acceptance of languages, but also occasionally with the *computation of string functions.* Suppose that f is a function from $(\Sigma - \{\sqcup\})^*$ to Σ^*, and let M be a Turing machine with alphabet Σ. We say that M *computes* f if, for any string $x \in (\Sigma - \{\sqcup\})^*$, $M(x) = f(x)$. If such an M exists, f is called a *recursive function.* □

For example, the function f from $\{0,1\}^*$ to $\{0,1,\sqcup\}^*$ such that $f(x) = \sqcup x$ is recursive since it can be computed by the machine in Figure 2.1. Also, the function s from $0\{0,1\}^*$ to $\{0,1,\sqcup\}^*$, such that $s(x)$ represents the integer $n+1$ in binary, where x is the binary representation of the positive integer n, is recursive: It can be computed by the Turing machine in Figure 2.2.

The rather unexpected terms for the various classes of string-related problems that can be solved by Turing machines ("recursive," "recursively enumerable") derive from the rich and involved history of the subject. Implicit in them is the main point of this chapter, namely the surprising computational power of Turing machines. These terms suggest that Turing machines are equivalent in power with arbitrarily general ("recursive") computer programs. Proposition 3.5 in the next chapter provides a further explanation for the term "recursively enumerable."

Thus, Turing machines can be thought of as algorithms for solving string-related problems. But how about our original project, to develop a notation for algorithms capable of attacking problems like those identified in the previous chapter, whose instances are mathematical objects such as graphs, networks, and numbers? To solve such a problem by a Turing machine, we must decide how to *represent by a string* an instance of the problem. Once we have fixed this representation, an algorithm for a decision problem is simply a Turing machine that decides the corresponding language. That is, it accepts if the input represents a "yes" instance of the problem, and rejects otherwise. Similarly, problems that require more complex output, such as MAX FLOW, are solved by the Turing machine that computes the appropriate function from strings to

strings (where the output is similarly represented as a string).

It should be clear that this proposal is quite general. Any "finite" mathematical object of interest can be represented by a finite string over an appropriate alphabet. For example, elements of finite sets, such as the nodes of a graph, can be represented as integers in binary. Pairs and k-tuples of simpler mathematical objects are represented by using parentheses and commas. Finite sets of simpler objects are represented by using set brackets, and so on. Or, perhaps, a graph can be represented by its *adjacency matrix*, which in turn can be arranged as a string, with rows separated by some special symbol such as ';' (see Figure 2.4).

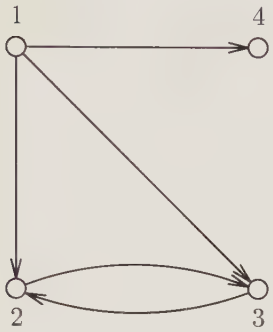
There is a wide range of acceptable representations of integers, finite sets, graphs, and other such elementary objects. They may differ a lot in form and succinctness. *However, all acceptable encodings are related polynomially.* That is, if A and B are both "reasonable" representations of the same set of instances, and representation A of an instance is a string with n symbols, then representation B of the same instance has length at most $p(n)$, for some polynomial p. For example, representing a graph with no isolated points by its adjacency matrix is at most quadratically more wasteful than representing it by an adjacency list (recall Figure 2.4).

Representing numbers in unary, instead of binary or decimal, is about the only possible slip in this regard. Obviously, the unary representation (in which, for example, number 14 is "IIIIIIIIIIIII") requires exponentially more symbols than the binary representation. As a result, the complexity of an algorithm, measured as a function of the length of the input of the Turing machine, may seem deceptively favorable (recall the analysis of our first attempt at an algorithm for MAX FLOW in Section 1.2). In the course of this book, when we discuss a Turing machine that solves a particular computational problem, we shall always assume that a reasonably succinct input representation, such as the ones in Figure 2.4, is used. In particular, *numbers will always be represented in binary.*

Complexity theory is quite independent of the input representation issue. It could have been developed in the isolation of strings and languages. But a sensible choice of representation (which in practice means avoiding the unary notation for numbers) makes the results of complexity theory immediately relevant to actual problems and computational practice. A valuable reward is the surprisingly close identification of complexity-theoretic concepts with computational problems, which is perhaps the main theme of this book.

2.3 TURING MACHINES WITH MULTIPLE STRINGS

We must next define formally the time and space expended by Turing machine computations. This is best done by first introducing a generalization of the Turing machine, the *Turing machine with multiple strings* and associated cursors.



"{(1, 100), (1, 10), (10, 11), (1, 11), (11, 10)}"

"(0111; 0010; 0100; 0000)"

Figure 2-4. Graph and representations.

We shall show that any such device can be simulated, with an inconsequential loss of efficiency, by an ordinary Turing machine, and so this feature is no departure from our commitment to Turing machines as our basic model. This result will also be a first important advance toward our implicit goal in this chapter, which is to demonstrate convincingly the surprising power of Turing machines.

Definition 2.4: A *k-string Turing machine*, where $k \geq 1$ is an integer, is a quadruple $M = (K, \Sigma, \delta, s)$, where K, Σ, and s are exactly as in ordinary Turing machines. δ is a program that must reflect the complexities of multiple strings. Intuitively, δ decides the next state as before, but also decides *for each string* the symbol overwritten, and the direction of cursor motion by looking at the current state and the current symbol *at each string*. Formally, δ is a function from $K \times \Sigma^k$ to $(K \cup \{h, \text{"yes"}, \text{"no"}\}) \times (\Sigma \times \{\leftarrow, \rightarrow, -\})^k$. Intuitively, $\delta(q, \sigma_1, \ldots, \sigma_k) = (p, \rho_1, D_1, \ldots, \rho_k, D_k)$ means that, if M is in state q, the cursor of the first string is scanning a σ_1, that of the second a σ_2, and so on, then the next state will be p, the first cursor will write ρ_1 and move in the direction indicated by D_1, and so on for the other cursors. $\triangleright$ still cannot be overwritten or passed on to the left: If $\sigma_i = \triangleright$, then $\rho_i = \triangleright$ and $D_i = \rightarrow$. Initially, all strings start with a $\triangleright$; the first string also contains the input. The outcome of the computation of a k-string machine M on input x is as with ordinary machines, with one difference: In the case of machines that compute functions, the output can be read from the *last* (kth) string when the machine halts. □

Example 2.4: We can decide palindromes more efficiently than in Example 2.3, using the 2-string Turing machine shown in Figure 2.5. This machine starts by copying its input in the second string. Next, it positions the cursor of the first string at the first symbol of the input, and the cursor of the second string at the last symbol of the copy. Then, it moves the two cursors in opposite

$p \in K,$	$\sigma_1 \in \Sigma$	$\sigma_2 \in \Sigma$	$\delta(p, \sigma_1, \sigma_2)$
$s,$	0	$\sqcup$	$(s, 0, \rightarrow, 0, \rightarrow)$
$s,$	1	$\sqcup$	$(s, 1, \rightarrow, 1, \rightarrow)$
$s,$	$\triangleright$	$\triangleright$	$(s, \triangleright, \rightarrow, \triangleright, \rightarrow)$
$s,$	$\sqcup$	$\sqcup$	$(q, \sqcup, \leftarrow, \sqcup, -)$
$q,$	0	$\sqcup$	$(q, 0, \leftarrow, \sqcup, -)$
$q,$	1	$\sqcup$	$(q, 1, \leftarrow, \sqcup, -)$
$q,$	$\triangleright$	$\sqcup$	$(p, \triangleright, \rightarrow, \sqcup, \leftarrow)$
$p,$	0	0	$(p, 0, \rightarrow, \sqcup, \leftarrow)$
$p,$	1	1	$(p, 1, \rightarrow, \sqcup, \leftarrow)$
$p,$	0	1	("no", $0, -, 1, -$)
$p,$	1	0	("no", $1, -, 0, -$)
$p,$	$\sqcup$	$\triangleright$	("yes", $\sqcup, -, \triangleright, \rightarrow$)

Figure 2.5. 2-string Turing machine for palindromes.

directions, checking that the two symbols under them are identical at all steps, at the same time erasing the copy. □

A *configuration* of a k-string Turing machine is defined analogously with ordinary Turing machines. It is a $(2k+1)$-tuple $(q, w_1, u_1, \ldots w_k, u_k)$, where q is the current state, the ith string reads $w_i u_i$, and the last symbol of w_i is holding the ith cursor. We say that $(q, w_1, u_1, \ldots, w_k, u_k)$ *yields in one step* $(q', w_1', u_1', \ldots, u_k' w_k')$ (denoted $(q, w_1, u_1, \ldots, w_k, u_k) \overset{M}{\rightarrow} (q', w_1', u_1', \ldots, w_k, u_k')$) if the following is true. First, suppose that σ_i is the last symbol of w_i, for $i = 1, \ldots, k$, and suppose that $\delta(q, \sigma_1, \ldots, \sigma_k) = (p, \rho_1, D_1, \ldots, \rho_k, D_k)$. Then, for $i = 1, \ldots, k$ we have the following: If $D_i = \rightarrow$, then w_i' is w_i with its last symbol (which was a σ_i) replaced by ρ_i, and the first symbol of u_i appended to it ($\sqcup$ if u_i is the empty string); u_i' is u_i with the first symbol removed (or, if u_i was the empty string, u_i' remains empty). If $D_i = \leftarrow$, then w_i' is w_i with σ_i omitted from its end, and u_i' is u_i with ρ_i attached in the beginning. Finally, if $D_i = -$, then w_i' is w_i with the ending σ_i replaced by ρ_i, and $u_i' = u_i$. In other words, the conditions for yielding in single-string Turing machines must hold at each string. The relations "yields in n steps" and plain "yields" are defined analogously with ordinary Turing machines.

A k-string Turing machine starts its computation on input x with the configuration $(s, \triangleright, x, \triangleright, \epsilon \ldots, \triangleright, \epsilon)$; that is, the input is the first string, and all strings start with an $\triangleright$. If $(s, \triangleright, x, \triangleright, \epsilon \ldots, \triangleright, \epsilon) \overset{M^*}{\rightarrow}$ ("yes", $w_1, u_1 \ldots, w_k, u_k$), for some strings $w_1, u_1, \ldots, u_k$, then we say that $M(x) =$ "yes"; if $(s, \triangleright, x, \triangleright, \epsilon \ldots, \triangleright, \epsilon) \overset{M^*}{\rightarrow}$ ("no", $w_1, u_1 \ldots, w_k, u_k$) then we say that $M(x) =$ "no". Finally, if the machine halts at configuration $(h, u_1, u_1 \ldots, w_k, u_k)$, then $M(x) = y$, where y is $w_k u_k$

with the leading ▷ and all trailing ␣s removed. That is, if M halts at state h (the state that signals the output is ready), the output of the computation is contained in the last string. Notice that, by these conventions, an ordinary Turing machine is indeed a k-string Turing machine with $k = 1$. Also, once the meaning of $M(x)$ has been defined, we can simply extend to multistring Turing machines the definitions of function computation, and language decision and acceptance of the previous section.

Definition 2.5: We shall use the multistring model of the Turing machine as the basis of our notion of the time expended by Turing machine computations (for space we shall need a minor modification introduced later in this chapter). If for a k-string Turing machine M and input x we have $(s, \triangleright, x, \triangleright, \epsilon, \ldots, \triangleright, \epsilon) \xrightarrow{M}{}^{t} (H, w_1, u_1 \ldots, w_s, u_s)$ for some $H \in \{h, \text{"yes"}, \text{"no"}\}$, then the *time required by M on input x* is t. That is, the time required is simply the number of steps to halting. If $M(x) = \nearrow$, then the time required by M on x is thought to be ∞ (this will rarely be the case in this book).

Defining the time requirements of a single computation is only the start. What we really need is a notion that reflects our interest in solving *any* instance of a problem, instead of isolated instances. Recall that the performance of algorithms in the previous chapter was characterized by the amount of time and space required on instances of "size" n, *when these amounts were expressed as a function of* n. For graphs, we used the number of nodes as a measure of "size." For strings, the natural measure of size is the length of the string. Accordingly, let f be a function from the nonnegative integers to the nonnegative integers. We say that machine M *operates within time* $f(n)$ if, for any input string x, the time required by M on x is at most $f(|x|)$ (by $|x|$ we denote the *length* of string x). Function $f(n)$ is a *time bound* for M.

Suppose now that a language $L \subset (\Sigma - \{\sqcup\})^*$ is decided by a multistring Turing machine operating in time $f(n)$. We say that $L \in \mathbf{TIME}(f(n))$. That is, $\mathbf{TIME}(f(n))$ is a set of languages. It contains exactly those languages that can be decided by Turing machines with multiple strings operating within the time bound $f(n)$. □

$\mathbf{TIME}(f(n))$ is what we call a *complexity class*. It is a set of languages (hopefully, including many that represent important decision problems). The property shared by these languages is that they can all be decided within some specified bound on some aspect of their performance (time, soon space, and later others). Complexity classes, and their relationship with the problems they contain, are the main objects of study in this book.

Example 2.5: It is easy to calculate the number of steps needed by the machine in Figure 2.3 for deciding whether a string of length n is a palindrome. That machine operates in $\lceil \frac{n}{2} \rceil$ stages. In the first stage in $2n + 1$ steps we compare the first and the last symbol, then the process is repeated with a string of length

$n-2$, then with one of length $n-4$, and so on. The total number of steps is at most $(2n+1)+(2n-3)+\ldots$, which is $f(n)=\frac{(n+1)(n+2)}{2}$. Therefore, the language of all palindromes is in $\mathbf{TIME}(\frac{(n+1)(n+2)}{2})$. We shall soon see that what is worth remembering from this exercise is that $f(n)$ is quadratic in n; that is, we only care that $f(n)=\mathcal{O}(n^2)$.

Naturally, this is our most pessimistic estimate. For a string like 01^n, the machine will detect in $n+3$ steps that it is not a palindrome, and halt. But in determining $f(n)$ we must take into account the *worst-case inputs* (which happen to be the strings that are actual palindromes).

In fact, it can be shown that *any* single-string Turing machine for deciding palindromes must take time $\Omega(n^2)$ (see Problems 2.8.4 and 2.8.5). Interestingly, the 2-string Turing machine of Example 2.4 takes time at most $f'(n)=3n+3=\mathcal{O}(n)$, establishing the the palindromes are in $\mathbf{TIME}(3n+3)$. □

The last paragraph implies that, by employing multistring Turing machines, we cabn achieve quadratic savings in time. We next prove that the savings can never be more than quadratic:

Theorem 2.1: Given any k-string Turing machine M operating within time $f(n)$, we can construct a Turing machine M' operating within time $\mathcal{O}(f(n)^2)$ and such that, for any input x, $M(x)=M'(x)$.

Proof: Suppose that $M=(K,\Sigma,\delta,s)$; we shall describe $M'=(K',\Sigma',\delta',s)$. M''s single string must "simulate" the k strings of M. One way to do this would be to maintain in M''s string the *concatenation* of the strings of M (without, of course, the $\triangleright$s that would come in the way of going back and forth in the string). We must also "remember" the position of each cursor, as well as the current right end of each string.

To accomplish all this, we let $\Sigma'=\Sigma\cup\underline{\Sigma}\cup\{\triangleright',\triangleleft\}$. Here $\underline{\Sigma}=\{\underline{\sigma}:\sigma\in\Sigma\}$ is a set of *cursor* versions of the symbols in Σ. $\triangleright'$ is a new version of $\triangleright$ which can be passed over to the left, and $\triangleleft$ marks the right end of a string. Thus, any configuration $(q,w_1,u_1\ldots,w_k,u_k)$ can be simulated by the configuration of M' $(q,\triangleright,w_1'u_1\triangleleft w_2'u_2\triangleleft\ldots w_k'u_k\triangleleft\triangleleft)$. Here w_i' is w_i with the leading $\triangleright$ replaced by $\triangleright'$, and the last symbol σ_i by $\underline{\sigma_i}$. The last two $\triangleleft$s signal the end of M''s string.

For the simulation to begin, M' has simply to shift its input one position to the right, precede it with a $\triangleright'$, and write the string $\triangleleft(\triangleright'\triangleleft)^{k-1}\triangleleft$ after its input. This can be easily accomplished by adding to the states of M' $2k+2$ new states, whose sole purpose is to perform this writing.

To simulate a move of M, M' scans twice its string from left to right and back. In the first scan, M' gathers information concerning the k currently scanned symbols in M: They are the k underlined symbols encountered. To do this "remembering," M' must contain new states, each of which corresponds to a particular combination of a state of M and of a k-tuple of symbols of M.

Based on its state at the end of the first scan, M' knows what changes

need to be performed on the string to reflect changes in the strings of M at the move being simulated. Then M' scans its string again from left to right, stopping at each underlined symbol to rewrite one or two symbols nearby, in a manner that reflects the symbols overwritten and the cursor motions of M at this string during this move. These updates can be easily performed based on the information available to M.

There is however one complication: If a cursor scanning the right end of a string needs to move right, we must create space for a new symbol (a $\sqcup$). This is done by first marking the currently scanned $\triangleleft$ by a special mark, making it $\triangleleft'$, moving M''s cursor all the way to the $\triangleleft\triangleleft$ on the right end, and moving all symbols one position to the right, as was done by the Turing machine in Example 2.1. When the $\triangleleft'$ is encountered, it is moved to the right as a simple $\triangleleft$, and a $\sqcup$ is overwritten in its old position. Then we go on to implement the changes in the next string of M.

The simulation proceeds until M halts. At this point, M' erases all strings of M except the last (so that its output is the same as that of M) and halts.

How long does the operation of M' on an input x take? Since M halts within time $f(|x|)$, during its operation none of its strings ever becomes longer than $f(|x|)$ (this is a very basic fact about any reasonable model of computation: It cannot waste more space than time!). Thus the total length of the string of M' is never more than $k(f(|x|)+1)+1$ (to account for the $\triangleleft$s). Simulating a move thus takes at most two traversals of this string from left to right and back $(4k(f(|x|)+1)+4$ steps), plus at most $3k(f(|x|)+1)+3$ steps per each string of M simulated. The total is $\mathcal{O}(k^2 f(|x|)^2)$, or, since k is fixed and independent of x, $\mathcal{O}(f(|x|)^2)$. $\square$

The quadratic dependence on the number of strings can be avoided by a different simulation (see Problem 2.8.6); but the quadratic dependence on f cannot be removed (recall the last paragraph of Example 2.5, and see Problems 2.8.4 and 2.8.5).

Theorem 2.1 is strong evidence of the power and stability of Turing machines as a model of computation: Adding a bounded number of strings does not increase their computational capabilities, and affects their efficiency only polynomially. We shall see in Section 2.5 that the addition of much stronger features, very much reminiscent of actual computers, also fails to significantly alter the capabilities of the basic Turing machine model. The eventual thesis implied by these and other similar results is that *there is no conceivable "realistic" improvement on the Turing machine that will increase the domain of the languages such machines decide, or will affect their speed more than polynomially.*

Time complexity.

generally hold belief.

cannot

→ Computability Church-Turing Thesis.

2.4 LINEAR SPEEDUP

Recall that, in estimating the time performance of the algorithms in the last chapter, we used the $\mathcal{O}$ notation a lot. This reflects our limited interest in the precise multiplicative and additive constants in our time and space bounds. Constants are, of course, of great interest to computer science. Unfortunately, it does not seem possible to build an elegant theory that accounts for them†. Furthermore, advances in hardware have in the past improved the performance of computers so rapidly, that algorithm designers can compete only by improving the *rate of growth* of the time requirements of their algorithms.

Our first theorem on the complexity of Turing machine computations essentially states exactly this, that multiplicative constants are not important, once we adopt the k-string Turing machine as our vehicle for defining time complexity (a companion result, Theorem 7.1 in Section 7.2, will establish that rates of growth *do* make a difference).

Theorem 2.2: Let $L \in \mathbf{TIME}(f(n))$. Then, for any $\epsilon > 0$, $L \in \mathbf{TIME}(f'(n))$, where $f'(n) = \epsilon f(n) + n + 2$.

Proof: Let $M = (K, \Sigma, \delta, s)$ be a k-string Turing machine that decides L and operates in time $f(n)$. We shall construct a k'-string Turing machine $M' = (K', \Sigma', \delta', s')$ operating within the time bound $f'(n)$, and which *simulates* M. The number k' of strings of M' will be equal to k if $k > 1$, and it will be 2 if $k = 1$. The big savings in the time performance comes from a simple trick (akin to increasing the word length of a computer): Each symbol of M' encodes several symbols of M. As a result, M' can simulate several moves of M by one move of its own. The linear term comes from an initial phase, in which the input of M is condensed for further processing by M'.

To fill in the details, let m be an integer, depending on M and ϵ alone, that will be fixed later. The alphabet of M' will contain, besides those of M, *all m-tuples of symbols of M*. That is, $\Sigma' = \Sigma \cup \Sigma^m$. In the beginning, M' moves the cursor in the first string to the right, effectively reading the input, x. When m symbols have been read, say $\sigma_1\sigma_2\ldots\sigma_m$, then the *single symbol* $(\sigma_1, \sigma_2, \ldots, \sigma_m) \in \Sigma'$ is printed in the second string. This is easy to do, by "remembering" in the state of M' the symbols read (just like the Turing machine in Example 2.3). Remembering is achieved by adding to K' the states in $K \times \Sigma^i$, $i = 1, \ldots, m-1$. If the end of the input is encountered in the middle of building an m-tuple (that is, if $|x|$ is not a multiple of m), the m-tuple is "padded" by the right number of $\sqcup$s to the right. After $m\lceil \frac{|x|}{m} \rceil + 2$ steps the second string contains (after its leftmost $\triangleright$) a condensed version of the input, as required. All other strings, if any, have remained $\triangleright$.

---

† Let us not forget that, in the interest of simplicity and elegance, we even belittle *polynomial differences* in complexity; in other words, we disregard constants *in the exponent*...

From now on we treat the second string as the one containing the input; if $k > 1$ then we use the first string as an ordinary work string (we write a $\triangleright$ after the end of the input so that we will never revisit that part). All k strings will henceforth contain exclusively m-tuples of symbols.

The main part of the simulation of M by M' starts now. M' repeatedly simulates m steps of M by six or fewer steps of its own. We call such a simulation of m steps of M by M' a *stage*. At the beginning of each stage, the state of M' contains a $(k+1)$-tuple $(q, j_1, \ldots, j_k)$. q is the state of M at the beginning of the stage, and $j_i \leq m$ is the precise position of the ith cursor *within* the m-tuple scanned. That is, if at some point of the simulation the state of M is q and the ith cursor of M is at the ℓth symbol after the first $\triangleright$, then the $i+1$st cursor of M' will point to the $\lceil \frac{\ell}{m} \rceil$th symbol to the right of $\triangleright$, and the state of M' will be $(q, \ell \bmod m)$.

Then (and this is the main point of the simulation) M' moves all of its cursors first to the left by one position, then to the right twice, and then again to the left once. The state of M' now "remembers" all symbols at or next to all cursors (this requires to add to K' all states in $K \times \{1, \ldots, m\} \times \Sigma^{3mk}$). The important thing to notice is that *based on this information, M' can fully predict the next m moves of M*. The reason is that, in m moves, no cursor of M can get out of the current m-tuple and the ones to its left and right. Hence, M' can now use its δ' function to implement in two steps the changes in string contents and state brought about by the next m moves of M (two steps are enough because these changes are confined in the current m-tuple and possibly one of its two neighbors). The simulation of m steps of M is complete in only six steps of M'.

Once M accepts or rejects its input, M' also halts and does the same. The total time spend on input x by M' is $|x| + 2 + 6\lceil \frac{f(|x|)}{m} \rceil$ at worst. Choosing $m = \lceil \frac{6}{\epsilon} \rceil$, we have the theorem. □

It is amusing to note that our construction for proving Theorem 2.2 illustrates a point we made earlier, namely that advances in hardware make constants meaningless. Multiplying the number of states (let alone the number of symbols) by $m|\Sigma|^{3ms}$ is some advance in hardware indeed!

But let us consider the significance of Theorem 2.2. Machines that operate in time bounds smaller than n clearly exist, but they are of no real interest[‡], since a Turing machine needs n steps just to read all of its input. So, any "decent" time bound will obey $f(n) \geq n$. If $f(n)$ is linear, something like cn for some $c > 1$, then the theorem says that the constant c can be made arbitrarily close to 1. If $f(n)$ is superlinear, say $14n^2 + 31n$, then the constant in the leading term (14 in this example) can become arbitrarily small, that is, the

[‡] In contrast, machines that operate in *space* less than n are of great interest indeed. Recall the last remark in Section 1.2, and see the next section.

time bound can suffer *an arbitrary linear speedup*. The lower-order terms, like $31n$ above, can be completely discarded. Thus, we shall henceforth feel justified to use freely the $\mathcal{O}$ notation in our time bounds. Also, any polynomial time bound will be henceforth represented by its leading term (n^k for some integer $k \geq 1$). That is, if L is a polynomially decidable language, it is in $\mathbf{TIME}(n^k)$ for some integer $k > 0$. *The union of all these classes, that is, the set of all languages decidable in polynomial time by Turing machines, is denoted by* $\mathbf{P}$. The reader already knows that it is a most central complexity class.

2.5 SPACE BOUNDS

At first it seems straight-forward to define the space used by the computation $(s, \triangleright, \epsilon, \ldots, \triangleright, \epsilon) \overset{M^*}{\to} (H, w_1, u_1, \ldots, w_k, u_k)$ of a Turing machine with k strings. Since the strings cannot become shorter in our model, the lengths of the final strings $w_i u_i$, $i = 1, \ldots, k$ capture the amount of storage required by each during the computation. We can either add up these lengths, or take their maximum. There are arguments in favor of both approaches, but we shall cut the discussion short by noting that the two estimates differ by at most k, *a constant*. Let us say for the time being that the *space used by the machine* M *on* x is $\sum_{i=1}^{k} |w_i u_i|$. *It turns out that this estimate represents a serious overcharge*. Consider the following example:

Example 2.6: Can we recognize palindromes in space substantially less than n? In view of our definition of the space used by a computation, the sum of the lengths of the strings at halting, this is impossible: One of these lengths, namely that of the first string carrying the input, will always be at least $|x|+1$.

But consider the following 3-string Turing machine that recognizes palindromes: The first string contains the input, *and is never overwritten*. The machine works in stages. At the ith stage the second string contains integer i in binary. During the ith stage we try to identify and remember the ith symbol of x. We do this by initializing the third string to $j = 1$, and then comparing i and j. Comparisons of the contents of the two different strings are easy to do by moving the two cursors. If $j < i$, then we increment j in binary (recall the machine in Example 2.2), and advance the first cursor to the right, to inspect the next input symbol. If $i = j$ we remember in our state the currently scanned symbol of the input, we reinitialize j to 1, and try to identify the ith from the last symbol of x. This is done much in the same way, with the first cursor now marching from right to left instead from left to right.

If the two symbols are unequal, we halt with "no". If they are equal, we increment i by one, and begin the next stage. If, finally, during a stage, the alleged jth symbol of x turns out to be a $\sqcup$, this means that $i > n$, and the input is a palindrome. We halt with a "yes".

How much space is needed for the operation of this machine? It seems

fair to assess that the space used up by this machine is $\mathcal{O}(\log n)$. The machine surely looks at the input (computation would be impossible otherwise) *but in a read-only fashion.* There is no writing on the first string. The other two strings are kept at most $\log n + 1$ long. □

This example leads us to the following definition: Let $k > 2$ be an integer. A *k-string Turing machine with input and output* is an an ordinary k-string Turing machine, with one important restriction on the program δ: Whenever $\delta(q, \sigma_1, \ldots, \sigma_s) = (p, \rho_1, D_1, \ldots, \rho_k, D_k)$, then (a) $\rho_1 = \sigma_1$, and (b) $D_k \neq \leftarrow$. Furthermore, (c) if $\sigma_1 = \sqcup$ then $D_1 = \leftarrow$. Requirement (a) says that at each move, the symbol "written" on the first string is always the same as the old symbol, and hence the machine effectively has a read-only input string. Requirement (b) states that in the last (output) string the cursor never moves to the left, and hence the output string is effectively write-only. Finally, (c) guarantees that the cursor of the input string does not wander off into the $\sqcup$s after the end of the input. It is a useful technical requirement, but not necessary.

These restrictions lead to an accurate definition of space requirements, without "overcharging" the computation for reading the input and for providing output, that is, for functions involving no "remembering" of earlier results of the computation. With these restrictions we shall be able to study space bounds smaller than n. However, let us first observe that these restrictions do not change the capabilities of a Turing machine.

Proposition 2.2: For any k-string Turing machine M operating within time bound $f(n)$ there is a $(k+2)$-string Turing machine M' with input and output, which operates within time bound $\mathcal{O}(f(n))$.

Proof: Machine M' starts by copying its input on the second string, and then simulating the k strings of M on its strings numbered 2 through $k+1$. When M halts, M' copies its output to the $k+2$nd string, and halts. □

Definition 2.6: Suppose that, for a k-string Turing machine M and an input x $(s, \triangleright, x, \ldots, \triangleright, \epsilon) \overset{M^*}{\rightarrow} (H, w_1, u_1 \ldots, w_k, u_k)$, where $H \in \{h, \text{"yes"}, \text{"no"}\}$ is a halting state. Then the *space required by M on input x* is $\sum_{i=1}^{k} |w_i u_i|$. *If, however, M is a machine with input and output, then the space required by M on input x is* $\sum_{i=2}^{k-1} |w_i u_i|$. Suppose now that f is a function from $\mathbf{N}$ to $\mathbf{N}$. We say that *Turing machine M operates within space bound $f(n)$* if, for any input x, M requires space at most $f(|x|)$.

Finally, let L be a language. We say that L is in the space complexity class $\mathbf{SPACE}(f(n))$ if there is a Turing machine with input and output that decides L and operates within space bound $f(n)$. For example, palindromes were shown to be in the space complexity class $\mathbf{SPACE}(\log n)$. This important space complexity class is usually referred to as $\mathbf{L}$. □

We finish this section with an analog of Theorem 2.2, establishing that, for

space as well, constants don't count. Its proof is an easy variant of that proof (see Problem 2.8.14).

Theorem 2.3: Let L be a language in **SPACE**$(f(n))$. Then, for any $\epsilon > 0$, $L \in$ **SPACE**$(2 + \epsilon f(n))$. □

2.6 RANDOM ACCESS MACHINES

We have already seen a number of examples and results suggesting that Turing machines, despite their weak data structure and instruction set, are quite powerful. But can they really be used to implement arbitrary algorithms?

We claim that they can. We shall not be able to prove this claim rigorously, because the notion of an "arbitrary algorithm" is something we cannot define (in fact, the Turing machine is the closest that we shall come to defining it in this book). The classical version of this informal claim, known as *Church's Thesis*, asserts that any algorithm can be rendered as a Turing machine. It is supported by the experiences of mathematicians in the early twentieth century while trying to formalize the notion of an "arbitrary algorithm." Quite independently, and starting from vastly different points of view and backgrounds, they all came up with models of computation that were ultimately shown equivalent in computational power to the Turing machine (which was the entry by one of them, Alan M. Turing; see the references for more on this fascinating story).

In the decades since those pioneering investigations, computing practice and the concern of computer scientists about time bounds has led them to another widespread belief, which can be thought of as a quantitative strengthening of Church's Thesis. It states that *any reasonable attempt to model mathematically computer algorithms and their time performance is bound to end up with a model of computation and associated time cost that is equivalent to Turing machines within a polynomial.* We have already seen this principle at work, when we noticed that multistring Turing machines can be simulated by the original model with only a quadratic loss of time. In this section we will present another, even more persuasive argument of this sort: We shall define a model of computation that reflects quite accurately many aspects of actual computer algorithms, and then show that it too is equivalent to the Turing machine.

A *random access machine*, or *RAM*, is a computing device which, like the Turing machine, consists of a program acting on a data structure. A RAM's data structure is an array of *registers*, each capable of containing an *arbitrarily large integer*, possibly negative. RAM instructions resemble the instruction set of actual computers; see Figure 2.6.

Formally, a *RAM program* $\Pi = (\pi_1, \pi_2, \ldots, \pi_m)$ is a finite sequence of *instructions*, where each instruction π_i of one of the types shown in Figure 2.6. At each point during a RAM computation, Register i, where $i \geq 0$, contains an integer, possibly negative, denoted r_i, and instruction π_κ is executed. Here

Instruction	Operand	Semantics
READ	j	$r_0 := i_j$
READ	$\uparrow j$	$r_0 := i_{r_j}$
STORE	j	$r_j := r_0$
STORE	$\uparrow j$	$r_{r_j} := r_0$
LOAD	x	$r_0 := x$
ADD	x	$r_0 := r_0 + x$
SUB	x	$r_0 := r_0 - x$
HALF		$r_0 := \lfloor \frac{r_0}{2} \rfloor$
JUMP	j	$\kappa := j$
JPOS	j	**if** $r_0 > 0$ **then** $\kappa := j$
JZERO	j	**if** $r_0 = 0$ **then** $\kappa := j$
JNEG	j	**if** $r_0 < 0$ **then** $\kappa := j$
HALT		$\kappa := 0$

j is an integer. r_j are the current contents of Register j. i_j is the jth input. x stands for an operand of the form j, "$\uparrow j$", or "$= j$". Instructions READ and STORE take no operand "$= j$". The value of operand j is r_j; that of "$\uparrow j$" is r_{r_j}; and that of "$= j$" is j. κ is the program counter. All instructions change κ to $\kappa + 1$, unless explicitly stated otherwise.

Figure 2.6. RAM instructions and their semantics.

κ, the "program counter," is another evolving parameter of the execution. The execution of the instruction may result in a change in the contents of one of the registers, and a change in κ. The changes, that is, the *semantics* of the instructions, are those one would expect from an actual computer, and are summarized in Figure 2.6.

Notice the special role of Register 0: It is the *accumulator*, where all arithmetic and logical computation takes place. Also notice the three *modes of addressing*, that is, of identifying an operand: Some instructions use an *operand* x, where x may be any of three types: j, "$\uparrow j$", or "$= j$". If x is an integer j, then x identifies the contents of Register j; if x is "$\uparrow j$", then it stands for the contents of the register *whose index is contained* in Register j; and if x is "$= j$", this means the integer j itself. The number thus identified is then used in the execution of the instruction.

An instruction also changes κ. Except for the last five instructions of Figure 2.6, the new value of κ is simply $\kappa+1$, that is, the instruction to be executed next is the next one. The four "jump" instructions change κ in another way, often dependent on the contents of the accumulator. Finally, the HALT instruction stops the computation. We can think that any *semantically wrong* instruction, such as one that addresses Register -14, is also a HALT instruction. Initially all

registers are initialized to zero, and the first instruction is about to be executed.

The *input* of a RAM program is a finite sequence of integers, contained in a finite array of *input registers*, $I = (i_1, \ldots, i_n)$. Any integer in the input can be transferred to the accumulator by a READ instruction. Upon halting, the output of the computation is contained in the accumulator.

The RAM executes the first instruction and implements the changes dictated by it, then it executes instruction π_κ, where κ is the new value of the program counter, and so on, until it halts. RAM computations can be formalized by defining the RAM analog of a configuration. A *configuration of a RAM* is a pair $C = (\kappa, R)$, where κ is the instruction to be executed, and $R = \{(j_1, r_{j_1}), (j_2, r_{j_2}), \ldots, (j_k, r_{j_k})\}$ is a finite set of *register-value pairs*, intuitively showing the current values of all registers that have been changed so far in the computation (recall that all others are zero). The initial configuration is $(1, \emptyset)$.

Let us fix a RAM program Π and an input $I = (i_1, \ldots, i_n)$. Suppose that $C = (\kappa, R)$ and $C' = (\kappa', R')$ are configurations. We say that (κ, R) *yields in one step* (κ', R'), written $(\kappa, R) \overset{\Pi, I}{\rightarrow} (\kappa', R')$, if the following holds: κ' is the new value of κ after the execution of π_κ, the κth instruction of Π. R', on the other hand, is the same as R in the case the κth instruction is one of the last five instructions in Figure 2.6. In all other cases, some Register j has a new value x, computed as indicated in Figure 2.6. To obtain R' we delete from R any pair (j, x') that may exist there, and add to it (j, x). This completes the definition of the relation $\overset{\Pi, I}{\rightarrow}$. We can now, as always, define $\overset{\Pi, I^k}{\rightarrow}$ (*yields in k steps*) and $\overset{\Pi, I^*}{\rightarrow}$ (*yields*).

Let Π be a RAM program, let D be a set of finite sequences of integers, and let ϕ be a function from D to the integers. We say that Π *computes* ϕ if, for any $I \in D$, $(1, \emptyset) \overset{\Pi, I^*}{\rightarrow} (0, R)$, where $(0, \phi(I)) \in R$.

We next define the time requirements of RAM programs. Despite the fact that the arithmetic operations of RAMs involve arbitrarily large integers, we *shall still count each RAM operation as a single step.* This may seem too liberal, because an ADD operation, say, involving large integers would appear to "cost" as much as the *logarithm* of the integers involved. Indeed, with more complicated arithmetic instructions the unit cost assumption would lead us to a gross underestimate of the power of RAM programs (see the references). However, our instruction set (in particular, the absence of a primitive multiplication instruction) ensures that the unit cost assumption is a simplification that is otherwise inconsequential. The mathematical justification for this choice will come with Theorem 2.5, stating that RAM programs, even with this extremely liberal notion of time, can be simulated by Turing machines with only a polynomial loss of efficiency.

We do need, however, logarithms in order to account for the size of the input. For i an integer, let $\mathrm{b}(i)$ be its *binary representation*, with no redundant leading 0s, and with a minus sign in front, if negative. The *length* of integer i is $\ell(i) = |\mathrm{b}(i)|$. If $I = (i_1, \ldots, i_k)$ is a sequence of integers, its *length* is defined as $\ell(I) = \sum_{j=1}^{k} \ell(i_j)$.

Suppose now that a RAM program Π computes a function ϕ from a domain D to the integers. Let f be a function from the nonnegative integers to the nonnegative integers, and suppose that, for any $I \in D$, $(1, \emptyset) \stackrel{\Pi, I^k}{\rightarrow} (0, R)$, where $k \leq f(\ell(I))$. Then we say that Π *computes* ϕ *in time* $f(n)$. In other words, we express the time required by a RAM computation on an input as a function of the total length of the input.

```
1.   READ 1
2.   STORE 1    (Register 1 contains i_1; during the kth iteration,
3.   STORE 5      Register 5 contains i_1 2^k. Currently, k = 0.)
4.   READ 2
5.   STORE 2    (Register 2 contains i_2)
6.   HALF       (k is incremented, and kth iteration begins)
7.   STORE 3    (Register 3 contains ⌊i_2/2^k⌋)
8.   ADD 3
9.   SUB 2
10.  JZERO 14
11.  LOAD 4     (add Register 5 to Register 4
12.  ADD 5        only if the kth least significant bit of i_2 is zero)
13.  STORE 4    (Register 4 contains i_1 · (i_2 mod 2^k))
14.  LOAD 5
15.  ADD 5
16.  STORE 5    (see comment of instruction 3)
17.  LOAD 3
18.  JZERO 20   (if ⌊i_2/2^k⌋ = 0 we are done)
19.  JUMP 5     (if not, we repeat)
20.  LOAD 4     (the result)
21.  HALT
```

Figure 2.7. RAM program for multiplication.

Example 2.7: Conspicuous in Figure 2.6 is the absence of the *multiplication* instruction MPLY. This is no great loss, because the program in Figure 2.7 computes the function $\phi : \mathbf{N}^2 \mapsto \mathbf{N}$ where $\phi(i_1, i_2) = i_i \cdot i_2$. The method is ordinary binary multiplication, where the instruction HALF is used to recover the binary representation of i_2 (actually, our instruction set contains this unusual

instruction precisely for this use).

In more detail, the program repeats $\lceil \log i_2 \rceil$ times the iteration in instructions 5 through 19. At the beginning of the kth iteration (we start with iteration 0) Register 3 contains $\lfloor i_2/2^k \rfloor$, Register 5 contains $i_1 2^k$, and register 4 contains $i_1 \cdot (i_2 \bmod 2^k)$. Each iteration strives to maintain this invariant (the reader is welcome to verify this). At the end of the iteration, we test whether Register 3 contains 0. If it does, we are done and ready to output the contents of Register 4. Otherwise, the next iteration is started.

It is easy to see that this program computes the product of two numbers in $\mathcal{O}(n)$ time. Recall that by $\mathcal{O}(n)$ time we mean *a total number of instructions that is proportional to the logarithm of the integers contained in the input.* This follows from the number of iterations of the program in Figure 2.7, which is at most $\log i_2 \leq \ell(I)$; each iteration entails the execution of a constant number of instructions.

Notice that, once we have seen this program, we can freely use the instruction MPLY x in our RAM programs, where x is any operand (j, $\uparrow j$, or $= j$). This can be simulated by running the program of Figure 2.7 (but possibly at a totally different set of registers). A few more instructions would take care of the case of negative multiplicands. □

Example 2.8: Recall the REACHABILITY problem from Section 1.1. The RAM program in Figure 2.8 solves REACHABILITY. It assumes that the input to the problem is provided in $n^2 + 1$ input registers, where Input Register 1 contains the number n of nodes, and Input Register $n(i-1)+j+1$ contains the (i,j)th entry of the adjacency matrix of the graph, for $i, j = 1, \ldots, n$. We assume that we wish to determine whether there is a path from node 1 to node n. The output of the program is simply 1 if a path exists (if the instance is a "yes" instance) and 0 otherwise.

The program uses Registers 1 through 7 as "scratch" storage. Register 3 always contains n. Register 1 contains i (the node just taken off S, recall the algorithm in Section 1.1), and Register 2 contains j, where we are currently investigating whether j should be inserted in S. In Registers 9 through $n+8$ we maintain the *mark bits* of the nodes: One if the node is marked (has been in S at some time in the past, recall the algorithm in Section 1.1), zero if it is not marked. Register $8+i$ contains the mark bit of node i.

Set S is maintained as a *stack;* as a result, the search will be depth-first. Register $n+9$ points to the top of the stack, that is, to the register containing the last node that was added to the stack. If Register $n+9$ is ever found to point to a register containing a number bigger than n (that is, to itself), then we know the stack is empty. For convenience, Register 8 contains the integer $n+9$, the address of the pointer to the top of the stack.

```
1.   READ 1
2.   STORE 3        (Register 3 contains n)
3.   ADD = 9
4.   STORE 8        (Register 8 contains the address of Register n + 9,
5.   ADD = 1          where the address of the top of the stack is stored)
6.   STORE ↑ 8      (Initially, the stack will contain only one element, node 1)
7.   STORE 7
8.   LOAD = 1       (push node 1 to the top of the stack;
9.   STORE ↑ 7        this ends the initialization phase)
10.  LOAD ↑ 8       (an iteration begins here)
11.  STORE 6
12.  LOAD ↑ 6       (load the top of the stack;
13.  SUB 3            if it is a number bigger than n,
14.  JNEG 52          the stack is empty and we must reject)
15.  STORE 1        (Register 1 contains i)
16.  LOAD 6
17.  SUB = 1
18.  STORE ↑ 8      (the top of the stack goes down by one)
19.  LOAD = 1
20.  STORE 2        (Register 2 contains j, initially 1)
21.  LOAD 1
22.  SUB = 1
23.  MPLY 3         (the multiplication program, Figure 2.7, is used here)
24.  ADD 2          (compute (i − 1) · n + j, the address of the
25.  ADD = 1          (i, j)th element of the adjacency matrix)
26.  STORE 4
27.  READ ↑ 4       (the (i, j)th element is read in)
28.  JZERO 48       (if zero, nothing to do)
29.  LOAD 2
30.  SUB 3
31.  JZERO 54       (if j = n we must accept)
32.  LOAD 2
33.  ADD = 8        (compute the address of the mark bit of j)
34.  STORE 5
35.  LOAD ↑ 5
36.  JPOS 48        (if the mark bit of j is one, nothing to do)
37.  LOAD = 1
38.  STORE ↑ 5      (the mark bit of j is now one)
39.  LOAD ↑ 8       (continued next page...)
```

Figure 2.8. RAM program for REACHABILITY.

```
40.   ADD = 1
41.   STORE 7
42.   STORE ↑ 8      (the top of the stack goes up by one)
43.   LOAD 2
44.   STORE ↑ 7      (push j onto the stack)
45.   LOAD 2
46.   SUB 3
47.   JZERO 10       (if j = n, then a new iteration must begin)
48.   LOAD 2
49.   ADD = 1
50.   STORE 2
51.   JUMP 21        (otherwise, j is incremented, and we repeat)
52.   LOAD = 0
53.   HALT           (and reject)
54.   LOAD = 1
55.   HALT           (and accept)
```

Figure 2.8 (Continued). RAM program for REACHABILITY.

Undoubtedly, this program keeps little of the original simplicity and elegance of the search algorithm, but it is an accurate rendering of that algorithm (you are invited to check). The number of steps is $\mathcal{O}(n^2 \log n)$: The work done per each READ instruction (each probe of the adjacency relation) is at most a constant plus $\log n$ for the MPLY instruction, and each entry is probed only once. To express this as a function of the length of the input (as we should for RAM programs), the length of the input $\ell(I)$ is about $(n^2 + \log n)$, which is $\Theta(n^2)$. Thus, the program works in time $\mathcal{O}(\ell(I) \log \ell(I))$.

If we wish to implement breadth-first search, this can be accomplished by realizing a *queue* instead of a stack; it would only be a minor modification of the RAM program of Figure 2.8. □

As a further illustration of the power of RAM programs, the next theorem states the rather expected fact that any Turing machine can be easily simulated by a RAM program. Suppose that $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$ is the alphabet of the Turing machine. Then we let D_Σ be this set of finite sequences of integers: $D_\Sigma = \{(i_1, \ldots, i_n, 0) : n \geq 0, 1 \leq i_j \leq k, j = 1, \ldots, n\}$. If $L \subset (\Sigma - \{\sqcup\})^*$ is a language, define ϕ_L to be the function from D_Σ to $\{0, 1\}$ with $\phi_L(i_1, \ldots, i_n, 0) = 1$ if $\sigma_{i_1} \ldots \sigma_{i_n} \in L$, and 0 otherwise. In other words, computing ϕ_L is the RAM equivalent of deciding L. The last 0 in the input sequence helps the RAM program "sense" the end of the Turing machine's input.

Theorem 2.4: Suppose that language L is in **TIME**$(f(n))$. Then there is a RAM program which computes the function ϕ_L in time $\mathcal{O}(f(n))$.

Proof: Let $M = (K, \Sigma, \delta, s)$ be the Turing machine that decides L in time $f(n)$. Our RAM program starts by copying the input to Registers 4 through $n+3$ (Registers 1 and 2 will be needed for the simulation, and Register 3 contains j, where $\sigma_j = \triangleright$). The value 4 is stored to Register 1, which will henceforth point to the register which contains the currently scanned symbol.

From now on, the program simulates the steps of the Turing machine one by one. The program has a set of instructions simulating each state $q \in K$. This sequence of instructions consists of k subsequences, where $k = |\Sigma|$. The jth subsequence starts at, say, instruction number N_{q,σ_j}. Suppose that $\delta(q, \sigma_j) = (p, \sigma_\ell, D)$. The instructions of the jth subsequence are the following:

N_{q,σ_j}.	LOAD $\uparrow 1$	(fetch the symbol scanned by the cursor)
$N_{q,\sigma_j}+1$.	SUB $= j$	(subtract j, the number of the symbol tested)
$N_{q,\sigma_j}+2$.	JZERO $N_{q,\sigma_j}+4$	(if the symbol is σ_j, we have work to do)
$N_{q,\sigma_j}+3$.	JUMP $N_{q,\sigma_{j+1}}$	(otherwise, try next symbol)
$N_{q,\sigma_j}+4$.	LOAD $= \ell$	(σ_ℓ is the symbol written)
$N_{q,\sigma_j}+5$.	STORE $\uparrow 1$	(write σ_ℓ)
$N_{q,\sigma_j}+6$.	LOAD 1	(the position of the cursor)
$N_{q,\sigma_j}+7$.	ADD $= d$	(d is 1 if $D = \rightarrow$, -1 if $D = \leftarrow$, 0 if $D = -$)
$N_{q,\sigma_j}+8$.	STORE 1	
$N_{q,\sigma_j}+9$.	JUMP N_{p,σ_1}	(start simulation of state p)

Finally, the states "yes" and "no" are simulated by simple sequences of RAM instructions which load the constant 1 (respectively, 0) on the accumulator, and then HALT.

The time needed to execute the instructions simulating a single move (that is, a state) is a constant (depending on M). We need $\mathcal{O}(n)$ more instructions to copy the input; the theorem follows. □

Our next result is a little more surprising. It states that any RAM can be simulated by a Turing machine with only a polynomial loss of efficiency. This is despite the rather liberal notion of time for RAM programs that we are using (for example, one time unit for an arbitrarily long addition). First we must fix appropriate "input-output conventions" that will enable Turing machines to behave like RAM programs. Suppose that ϕ is a function from D to the integers, where D is a set of finite sequences of integers. The binary representation of a sequence $I = (i_1, \ldots, i_n)$, denoted $\mathrm{b}(I)$, is the string $\mathrm{b}(i_1); \ldots; \mathrm{b}(i_n)$. We say that a Turing machine M *computes* ϕ if, for any $I \in D$, $M(\mathrm{b}(I)) = \mathrm{b}(\phi(I))$.

Theorem 2.5: If a RAM program Π computes a function ϕ in time $f(n)$, then there is a 7-string Turing machine M which computes ϕ in time $\mathcal{O}(f(n)^3)$.

Proof: The first string of M is an input string, never overwritten. The second string contains a representation of R, the register contents; it consists of a sequence of strings of the form $\mathrm{b}(i) : \mathrm{b}(r_i)$, separated by ';'s and possibly by runs of blanks. The sequence ends with an endmarker, say $\triangleleft$. Each time the

value of Register i (including an accumulator's value) is updated, the previous $\mathrm{b}(i) : \mathrm{b}(r_i)$ pair is erased (␣s are printed over it), and a new pair is attached to the right end of the string. $\triangleleft$ is shifted to mark the new end of the string.

The states of M are subdivided into m groups, where m is the number of instructions in Π. Each group implements one instruction of Π. The necessary contents of registers are obtained by scanning the second string from left to right. At most two such scannings will be necessary (in the case of a "$\uparrow j$" operand). String three contains the current value of the program counter κ. The fourth string is used to hold the register address currently sought. Each time a $\mathrm{b}(i) : \mathrm{b}(r_i)$ pair is scanned, the contents of the fourth string are compared to $\mathrm{b}(i)$, and if they are identical, $\mathrm{b}(r_i)$ is fetched to the other strings for processing. All operations are carried out in the three remaining strings. In the case of the three arithmetic operations, two of these strings hold the operands, and the result is computed in the other one. The second string is updated (a pair is erased and one is added to the right) after the computation. Finally, the next state starts the execution of the appropriate next instruction. If a HALT instruction is reached, the contents of Register 0 are fetched from the second string and written on the seventh string, the output.

Let us calculate the time needed to simulate in this way $f(n)$ steps of Π on input I, of length n. We first need to show the following useful fact:

> **Claim:** After the tth step of a RAM program computation on input I, the contents of any register have length at most $t + \ell(I) + \ell(B)$, where B is the largest integer referred to in an instruction of Π.
>
> ***Proof:*** By induction on t. It is true before the first step, when $t = 0$. Suppose that it is true up to the tth step. There are several cases. If the instruction executed is a "jump" instruction or a HALT, no register contents are changed. If it is a LOAD or STORE instruction, the contents of the register that was modified were present at another register at the previous step, and the claim holds. If it is a READ instruction, the $\ell(I)$ part of the estimate guarantees the claim. Finally, suppose that it is an arithmetic instruction, say ADD. It involves the addition of two integers, i and j. Each is either contained in a register at the last step, or is explicitly mentioned in Π (as in ADD = 314159), in which case it is smaller than B. The length of the result is at most one plus the length of the longest operand, which, by induction, is at most $t - 1 + \ell(I) + \ell(B)$. The situation with SUB is identical, and with HALF even easier (the result is shorter than the operand). □

We now proceed with the proof of the time bound. We have to show that simulating an instruction of Π by M takes $\mathcal{O}(f(n)^2)$ time, where n is the length of the input. Decoding Π's current instruction and the constants it contains can be done in constant time. Fetching the value of the registers involved in the

execution of the instruction from the second string of M takes $\mathcal{O}(f(n)^2)$ time (the second string contains $\mathcal{O}(f(n))$ pairs, each of length $\mathcal{O}(f(n))$, by the claim, and searching it can be done in linear time). The computation of the result itself involves simple arithmetic functions (ADD, SUB, HALF) on integers of length $\mathcal{O}(f(n))$, and thus can be done in $\mathcal{O}(f(n))$ time. The proof is complete. □

2.7 NONDETERMINISTIC MACHINES

We shall now break our chain of "reasonable" models of computation that can simulate each other with only a polynomial loss of efficiency. We shall introduce an *unrealistic* model of computation, the nondeterministic Turing machine. And we shall show that it can be simulated by our other models with an *exponential* loss of efficiency. Whether this exponential loss is inherent or an artifact of our limited understanding of nondeterminism is the famous $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ problem, the central subject of this book.

A *nondeterministic Turing machine* is a quadruple $N = (K, \Sigma, \Delta, s)$, much like the ordinary Turing machine. K, Σ, and s are as before. Reflecting the fact that a nondeterministic Turing machine does not have a single, uniquely defined next action, but a *choice* between several next actions, Δ is no longer a function from $K \times \Sigma$ to $(K \cup \{h, \text{"yes"}, \text{"no"}\}) \times \Sigma \times \{\leftarrow, \rightarrow, -\}$, but instead a *relation* $\Delta \subset (K \times \Sigma) \times [(K \cup \{h, \text{"yes"}, \text{"no"}\}) \times \Sigma \times \{\leftarrow, \rightarrow, -\}]$. That is, for each state-symbol combination, there may be *more than one* appropriate next steps—or none at all.

Nondeterministic Turing machine configurations are exactly like configurations of deterministic machines, but "yields" is no longer a function; it is instead a *relation*. We say that configuration (q, w, u) of the nondeterministic Turing machine N *yields* configuration (q', w', u') *in one step*, denoted $(q, w, u) \stackrel{N}{\rightarrow} (q', w', u')$, intuitively if there is a rule in Δ that makes this a legal transition. Formally, it means that there is a move $((q, \sigma), (q', \rho, D)) \in \Delta$ such that, either (a) $D = \rightarrow$, and w' is w with its last symbol (which was a σ) replaced by ρ, and the first symbol of u appended to it ($\sqcup$ if u is the empty string), and u' is u with the first symbol removed (or, if u was the empty string, u' remains empty); or (b) $D = \leftarrow$, w' is w with σ omitted from its end, and u' is u with ρ attached in the beginning; or finally, (c) $D = -$, w' is w with the ending σ replaced by ρ, and $u' = u$. $\stackrel{N}{\rightarrow}^k$ and $\stackrel{N}{\rightarrow}^*$ can be defined as usual, but $\stackrel{N}{\rightarrow}^k$ is, once more, no longer a function.

What makes nondeterministic machines so different and powerful is the very weak "input-output behavior" we demand of them, that is, our very liberal notion of what it means for such a machine to "solve a problem." Let L be a language and N a nondeterministic Turing machine. We say that N *decides* L if for any $x \in \Sigma^*$, the following is true: $x \in L$ if and only if $(s, \triangleright, x) \stackrel{N}{\rightarrow}^*$

("yes", w, u) for some strings w and u.

This is the crucial definition that sets nondeterministic machines apart from other models. An input is accepted if there is *some* sequence of nondeterministic choices that results in "yes". Other choices may result in rejection; just one accepting computation is enough. The string is rejected only if no sequence of choices can lead to acceptance. This *asymmetry* in the way "yes" and "no" instances are treated in nondeterminism is very reminiscent of the similar asymmetry in the definition of acceptance by Turing machines (Section 2.2). We shall see more on this analogy between acceptance and nondeterminism (and the way it breaks down in certain crucial respects).

We say that nondeterministic Turing machine N decides language L *in time* $f(n)$, where f is a function from the nonnegative integers to the nonnegative integers, if N decides L, and, moreover for any $x \in \Sigma^*$, if $(s, \triangleright, x) \overset{N}{\rightarrow}{}^{k} (q, u, w)$, then $k \leq f(|x|)$. That is, we require that N, besides deciding L, does not have computation paths longer than $f(n)$, where n is the length of the input. Thus, we charge to a nondeterministic computation an amount of time that may be considered unrealistically small. The amount of time charged is the "depth" of the computational activity that is going on—see Figure 2.9. Obviously, the "total computational activity" generated can be exponentially bigger.



Figure 2-9. Nondeterministic computation.

The set of languages decided by nondeterministic Turing machines within time f is a new, important kind of complexity class, denoted **NTIME**($f(n)$). *A most important nondeterministic complexity class is* **NP**, *the union of all* **NTIME**(n^k). Notice immediately that $\mathbf{P} \subseteq \mathbf{NP}$; the reason is that deterministic

machines form a subclass of the nondeterministic ones: They are precisely those for which the relation Δ happens to be a function.

Example 2.9: Recall that we do not know whether the decision version of the traveling salesman problem, TSP (D) (recall Section 1.3), is in **P**. However, it is easy to see that TSP (D) is in **NP**, because it can be decided by a nondeterministic Turing machine in time $\mathcal{O}(n^2)$. This machine, on an input containing a representation of the TSP instance, goes on to write *an arbitrary sequence of symbols*, no longer than its input. When this writing stops, the machine goes back and checks to see whether the string written is the representation of a permutation of the cities, and, if so, whether the permutation is a tour with cost B or less. Both tasks can easily be carried out in $\mathcal{O}(n^2)$ time by using a second string (multistring nondeterministic Turing machines are straightforward to define; they can be simulated by single-string ones again with a quadratic slowdown). If the string indeed encodes a tour cheaper than B, the machine accepts; otherwise, it rejects (perhaps hoping that another choice of symbols, written at some other branch of the tree in Figure 2.9, will end up accepting).

This machine indeed decides TSP (D), because it accepts its input if and only if it encodes a "yes" instance of TSP (D). If the input is a "yes" instance, this means that there is a tour of the cities with cost B or less. Therefore, there will be a computation of this machine that "guesses" precisely this permutation, checks that its cost is indeed below B, and thus ends up accepting. It does not matter that other computations of the machine may have guessed very costly tours, or plain nonsense strings, and will therefore reject; a single accepting computation is all that is needed for the machine to accept. Conversely, if there are no tours of cost less than B, there is no danger that a computation will accept: All computations will discover that their guess is wrong, and will reject. Thus, according to our convention, the input will be rejected. □

It would be remarkable if we could somehow turn this nondeterministic idea into a viable, polynomial-time algorithm for TSP (D). Unfortunately, at present we know of only exponential general methods for turning a nondeterministic algorithm into a deterministic one. We describe the most obvious such method next.

Theorem 2.6: Suppose that language L is decided by a nondeterministic Turing machine N in time $f(n)$. Then it is decided by a 3-string deterministic Turing machine M in time $\mathcal{O}(c^{f(n)})$, where $c > 1$ is some constant depending on N.

Notice that, using complexity class notation, we can restate this result very succinctly:

$$\mathbf{NTIME}(f(n)) \subseteq \bigcup_{c>1} \mathbf{TIME}(c^{f(n)}).$$

Proof: Let $N = (K, \Sigma, \Delta, s)$. For each $(q, \sigma) \in K \times \Sigma$, consider the set of choices $C_{q,\sigma} = \{(q', \sigma', D) : ((q, \sigma), (q', \sigma', D)) \in \Delta\}$. $C_{q,\sigma}$ is a finite set. Let $d = \max_{q,\sigma} |C_{q,\sigma}|$ (d could be called the "degree of nondeterminism" of N), and assume that $d > 1$ (otherwise, the machine is deterministic, and there is nothing to do).

The basic idea for simulating N is this: Any computation of N is a sequence of nondeterministic choices. Now, any sequence of t nondeterministic choices by N is essentially a sequence of t integers in the range $0, 1, \ldots, d-1$. The simulating deterministic machine M considers all such sequences of choices, in order of increasing length, and simulates N on each (notice that we cannot simply consider the sequences of length $f(|x|)$, where x is the input, because M must operate with no knowledge of the bound $f(n)$). While considering sequence $(c_1, c_2, \ldots, c_t)$, M maintains the sequence on its second string. Then M simulates the actions that N would have taken *had N taken choice c_i at step i* for its first t steps. If these choices would have led N to halting with a "yes" (possibly earlier than the tth step), then M halts with a "yes" as well. Otherwise, M proceeds to the next sequence of choices. Generating the next sequence is akin to calculating the next integer in d-ary, and can be done easily (recall Example 2.2).

How can M detect that N rejects, with no knowledge of the bound $f(n)$? The solution is simple: If M simulates all sequences of choice of a particular length t, and finds among them *no continuing computation* (that is, all computations of length t end with halting, presumably with "no" or h), then M can conclude that N rejects its input.

The time required by M to complete the simulation is bounded from above by $\sum_{t=1}^{f(n)} d^t = \mathcal{O}(d^{f(n)+1})$ (the number of sequences it needs to go through) times the time required to generate and consider each sequence, and this latter cost is easily seen to be $\mathcal{O}(2^{f(n)})$. □

We shall also be very interested in the *space* used by nondeterministic Turing machines. To do this correctly for space bounds smaller than linear, we shall need to define a *nondeterministic Turing machine with input and output*. It is a straight-forward task (which we omit) to combine the two extensions of Turing machines into one.

Given an k-string nondeterministic Turing machine with input and output $N = (K, \Sigma, \Delta, s)$, we say that N decides language L within space $f(n)$ if N decides L and, moreover, for any $x \in (\Sigma - \{\sqcup\})^*$, if $(s, \triangleright, x, \ldots, \triangleright, \epsilon) \overset{N^*}{\rightarrow} (q, w_1, u_1, \ldots, w_s, u_s)$, then $\sum_{j=2}^{s-1} |w_i u_i| \leq f(|x|)$. That is, we require that N under no circumstances uses space in its "scratch" strings greater than function f in the input length. Notice that our definition does not even require N to halt on all computations.

Example 2.10: Consider REACHABILITY (Section 1.1). Our deterministic al-

gorithm in Section 1.1 required linear space. In Chapter 7 we shall describe a clever, space-efficient deterministic algorithm for REACHABILITY: It requires space $\mathcal{O}(\log^2 n)$. It turns out that with nondeterminism we can solve REACHABILITY within space $\mathcal{O}(\log n)$.

The method is simple: Use two strings besides the input. In one write the current node i in binary (initially node 1), and in the other "guess" an integer $j \leq n$ (intuitively, the next node in the alleged path). Then go on to check at the input that the (i, j)th bit of the adjacency matrix of the graph is indeed one. If it is not, halt and reject (no harm is done, as long as some other sequence of choices leads to acceptance). If it is one, then first check whether $j = n$, and if it is, halt and accept (that n was reached means that there is a path from 1 to n). If j is not n, copy it to the other string (erasing i), and repeat.

It is easy to check that (a) the nondeterministic Turing machine described solves the REACHABILITY problem, and (b) at all points the space used in the two scratch strings (and perhaps other counters used in looking up entries of the adjacency matrix) is $\mathcal{O}(\log n)$. Notice that there may be arbitrarily long, indeed infinite, computations; but they are all confined within space $\log n$, the important parameter here. (In Chapter 7 we shall see how to standardize space-bounded machines so that they always halt; in the present case we could achieve this by equipping the machine with a counter incremented with each new node considered, and have the machine halt with "no" when the counter reaches n.)

This example demonstrates the power of nondeterminism with respect to space: There is no known algorithm for REACHABILITY that takes less than $\log^2 n$ space, while the problem can be "solved" nondeterministically in $\log n$ space. However, the gap now is much less dramatic than the exponential chasm between determinism and nondeterminism we encountered with respect to time complexity. In fact, this gap is the largest one known —indeed, possible. And it is not surprising that it occurs with REACHABILITY: This problem captures nondeterminism with respect to space so accurately, that it will be one of our main tools for studying this aspect of nondeterminism in Chapter 7. □

The nondeterministic Turing machine is not a true model of computation. Unlike the Turing machine and the random access machine, it was not the result of an urge to define a rigorous mathematical model for the formidable phenomenon of computation that was being either envisioned or practiced. Nondeterminism is a central concept in complexity theory because of its affinity not so much with computation itself, but with *the applications of computation*, most notably logic, combinatorial optimization, and artificial intelligence. A great part of logic (see Chapters 4 through 6) deals with *proofs*, a fundamentally nondeterministic concept. A sentence "may try" all possible proofs, and it succeeds in becoming a theorem if *any one of them works*. Similarly with a sentence being satisfied by a model —another central concept in logic, see Chapters 4–6. On the other hand, a typical problem in artificial intelligence and combinatorial

optimization seeks one solution among exponentially many that has the lowest cost, or satisfies certain complicated constraints; a nondeterministic machine can search an exponentially large solution space quite effortlessly.

2.8 NOTES, REFERENCES, AND PROBLEMS

2.8.1 Turing machines were defined by Alan M. Turing in

- A. M. Turing "On computable numbers, with an application to the *Entscheidungsproblem*," *Proc. London Math. Society, 2*, 42, pp. 230–265, 1936. Also, 43, pp. 544–546, 1937.

This was one of several independent, equivalent, and essentially simultaneous formulations of computation that were proposed in the 1930's, motivated mainly by the decision problem in first-order logic (see Chapters 5 and 6). Post proposed a similar formalism in

- E. Post "Finite combinatory processes: Formulation I," *J. Symbolic Logic, 1*, pp. 103–105, 1936.

Kleene formalized a comprehensive class of recursively defined, and therefore computable, number-theoretic functions in

- S. C. Kleene "General recursive functions of natural numbers," *Mathematische Annalen, 112*, pp. 727–742, 1936.

Church invented another notation for algorithms, the *λ-calculus*; see

- A. Church "The calculi of lambda-conversion," *Annals of Mathematical Studies, 6*, Princeton Univ. Press, Princeton N.J., 1941.

Finally, another string-oriented algorithmic notation was proposed by A. Markov

- A. Markov "Theory of algorithms," *Trudy Math. Inst. V. A. Steklova, 42*, 1954. English translation: Israel Program for Scientific Translations, Jerusalem, 1961.

For more extensive introductions to Turing machines, the theory of computation, as well as brief treatments of complexity theory, see

- J. E. Hopcroft and J. D. Ullman *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading Massachusetts, 1979;
- H. R. Lewis, C. H. Papadimitriou *Elements of the Theory of Computation*, Prentice-Hall, Englewood Cliffs, 1981.

Finally, see

- P. van Emde Boas "Machine models and simulations," pp. 1 61 in *The Handbook of Theoretical Computer Science, vol. I: Algorithms and Complexity*, edited by J. van Leeuwen, MIT Press, Cambridge, Massachusetts, 1990

for a very comprehensive treatment of various models of computation and their equivalences.

2.8.2 Problem: Prove by induction on $|x|$ that the Turing machine M of Example 2.1 computes the function $f(x) = \sqcup x$.

2.8.3 The palindromic Greek sentence "*Νιψον ανομηματα μη μοναν οψιν*," meaning "Wash away my sins not just my face," was an epigram on a fountain in the Byzantine church of Aghia Sophia in Constantinople (now Istanbul, Turkey).

2.8.4 Kolmogorov complexity. In what sense is the string 011010110111001 more complex than 0101010101010101? Since computational complexity deals with *infinite languages*, and not finite strings, it provides no answer. There is an important theory of *descriptional* complexity, founded by A. N. Kolmogorov, that deals with such questions. See

- A. N. Kolmogorov "Three approaches to the quantitative definition of information," *Prob. of Information Transmission* 1, 1, pp. 1–7, 1965, and
- M. Li and P. M. B. Vitányi "Kolmogorov complexity and its applications," pp. 187–254 in *The Handbook of Theoretical Computer Science, vol. I: Algorithms and Complexity*, edited by J. van Leeuwen, MIT Press, Cambridge, Massachusetts, 1990

Let $\mu = (M_1, M_2, \ldots)$ be an encoding of all Turing machines in the alphabet $\{0,1\}$; it should be clear that such encodings abound; see Section 3.1 for a discussion of the issue. We can then define, for each string $x \in \{0,1\}^*$ its *Kolmogorov complexity with respect to* μ as

$$K_\mu(x) = \min\{|M_j| : M_j(\epsilon) = x\}.$$

That is, $K_\mu(x)$ is the length of the smallest "program" in our encoding that outputs the string x.

(a) Show that there is an encoding μ_0 of Turing machines such, that for any other encoding μ there is a constant c such that, for all x, $K_{\mu_0}(x) \leq K_{\mu'}(x) + c$. (Use a universal Turing machine, see the next chapter.)

In view of this result, we are justified to henceforth omit reference to μ, and speak of the Kolmogorov complexity of x, $K(x)$ (understanding that a constant may be involved).

(b) Show that for all x, $K(x) \leq |x|$.

(c) Show that for all n there are strings x of length n with $K(x) \geq n$.

In other words, there are *incompressible strings*.

2.8.5 Lower bound for palindromes. One can use Kolmogorov complexity to prove lower bounds. Suppose that M is a one-string Turing machine that decides the language of palindromes defined in Example 2.3. We shall show that it requires $\Omega(n^2)$ steps.

If M is a Turing machine, x an input, and $0 \leq i \leq |x|$, and suppose that $(s, \epsilon, x) \overset{M}{\to}{}^{t-1} (q, u\sigma, v) \overset{M}{\to} (q', u', v')$, where either (a) $|u| = i - 1$ and $|u'| = i + 1$, or (b) $|u| = |u'| = i$. We say that *at the tth step the computation of M on x crosses the ith string position with the pair* (q, σ). In case (a) it crosses *to the right*, in case (b) *to the left*. The *ith crossing sequence* of M on x is the sequence $S_i(M, x) = ((q_1, \sigma_1), \ldots, (q_m, \sigma_m))$, such that there are step numbers $t_1 < t_2 < \ldots < t_m$, and at step t_j with $1 \leq j \leq m$ there is a crossing at the ith position with pair (q_j, σ_j). Moreover there are no more crossings at i. Obviously, odd-numbered crossings are to the right, and even-numbered crossings are to the left.

Intuitively, "from the point of view" of the first i string positions, what is important is not so much the changes in the contents of the remaining positions during the

computation, but just the ith crossing sequence. The following argument is based on this intuition.

Let M be a single-string Turing machine deciding palindromes over $\{0,1\}$, and consider its computation on the palindrome $x0^n x^R$, where x is any string of length n, and x^R denotes the *reversal* of x (x written backwards). Suppose that the total number of steps is $T(x)$. Then there is an i, $n \leq i \leq 2n$, such that the ith crossing sequence S_i has length $m \leq \frac{T(x)}{n}$. Suppose that this crossing sequence is $((q_1, \sigma_1), \ldots, (q_m, \sigma_m))$.

We conclude that *x can be identified as the unique string of length n* which, when concatenated with 0^{i-n} and given as an input to M, will cause the machine to first cross the right end of the string with (q_1, σ_1); furthermore, if M were to cross to the left with (q_2, σ_2) (we are using the crossings to the left instead of the rest of the string), the next crossing will be (q_3, σ_3). And so on, up to (q_m, σ_m). Notice that indeed this description uniquely identifies x; because if another string y had the same property, then M would end up accepting $y0^n x^R$, not a palindrome.

(a) Show that the description in the previous paragraph can be rendered as a binary string of length at most $c_M \frac{T(x)}{n} + 2 \log n$, where c_M is a constant depending only on M.

(b) Argue that there is a Turing machine M' which, given the description above, will generate x.

(c) Conclude that the *Kolmogorov complexity* of x (recall Problem 2.8.4) is at most $c_M \frac{T(x)}{n} + 2 \log n + c_{M'}$, where $c_{M'}$ depends only on M'.

(d) Use the results in Problem 2.8.4(c) above to show that M takes time $\Omega(n^2)$.

Therefore, the simulation in Theorem 2.1 is *asymptotically optimal.*

2.8.6 Problem: Give another proof of Theorem 2.1 by representing the k strings not next to each other, but *on top of each other.* Analyze the efficiency of your simulation, and compare with the proof of Theorem 2.1. (A single symbol of the simulating machine will encode k symbols of the simulated one. Naturally, the cursors will be all over the place.)

2.8.7 Problem: Suppose that we have a Turing machine with an infinite *two-dimensional string* (blackboard?). There are now moves of the form $\uparrow$ and $\downarrow$, along with $\leftarrow$ and $\rightarrow$. The input is written initially to the right of the initial cursor position.

(a) Give a detailed definition of the transition function of such a machine. What is a configuration?

(b) Show that such a machine can be simulated by a 3-string Turing machine with a *quadratic* loss of efficiency.

2.8.8 Problem: Suppose that Turing machines can *delete and insert symbols* in their string, instead of only overwriting.

(a) Define carefully the transition function and the computation of such machines.

(b) Show that such a machine can be simulated by an ordinary Turing machine with a quadratic loss of efficiency.

2.8.9 Problem: Show that any k-string Turing machine operating within time $f(n)$

can be simulated by a 2-string Turing machine within time $f(n) \log f(n)$. (This is a clever simulation, from

- F. C. Hennie and R. E. Stearns "Two-tape simulation of multitape Turing machines," *J.ACM 13*, 4, pp. 533–546, 1966.

The trick is to keep all strings on top of one another in a single string—that is, a single symbol will encode k, as in Problem 2.8.6 above— so that the locations of all cursors coincide; the strings are unbounded in both directions, say. Cursor motions are simulated by moving blocks of symbols around, into spaces that were intentionally written sparsely; the sizes of the blocks form a geometric progression. The second string is only used for fast copying.)

This result is a useful substitute for Theorem 2.1, when a Turing machine with a fixed number of strings has to be constructed, as it achieves better time bounds (see Problem 7.4.8).

2.8.10 Call a Turing machine M *oblivious* if the position of its cursors at the tth step of its computation on input x *depends only on* t *and* $|x|$, not x itself.

Problem: Show that the simulating two-string machine above can be made oblivious. (Oblivious machines are useful because they are closer to more "static" models of computation, such as Boolean circuits; see Problem 11.5.22.)

2.8.11 Regular languages. Suppose that a language L is decided by a Turing machine with input and output that uses *constant space*.

Problem: (a) Show that there is an equivalent Turing machine deciding L *with only a read-only input string* (acceptance and rejection is signaled by states "yes" and "no", say).

Such a machine is called a *two-way finite automaton*. A (one-way) finite automaton is a "two-way" finite automaton with only $\rightarrow$ cursor moves (a "yes" or "no" state is entered when the first blank is encountered). The language decided by a one-way finite automaton is called *regular*.

(b) Show that the following language is regular:

$$L = \{x \in \{0,1\}^* : x \text{ contains at least two 0s but not two consecutive 0s}\}.$$

For more on regular languages see the book by Lewis and Papadimitriou cited above, as well as Problems 3.4.2 and 20.2.13.

(c) Let $L \subseteq \Sigma^*$ be a language, and define the following equivalence relation on Σ^*: $x \equiv_L y$ iff for all $z \in \Sigma^*$ $xz \in L$ iff $yz \in L$. Show that L is regular if and only if there are finitely many equivalence classes in $\equiv_L$. How many equivalence classes are there in the case of the language in (b) above?

(d) Show that if a language L is decided by a two-way finite automaton, then it is regular. (This is a quite subtle argument from

- J. C. Shepherdson "The reduction of two-way automata to one-way automata," *IBM J. of Res. and Dev., 3*, pp. 198–200, 1959.

Show that $x \equiv_L y$ if and only if x and y "effect the same behavior" on the two-way machine. Here by "behavior" one should understand a certain mapping from state to state, of which of course there are finitely many. Use part (c).)

We conclude that **SPACE**(k), where k is any integer, is precisely the class of regular languages.

(e) Show that, if L is infinite and regular, then there are $x, y, z \in \Sigma^*$ such that $y \neq \epsilon$, and $xy^iz \in L$ for all $i \geq 0$. (Since there are finitely many states in the machine, on a long enough string one must repeat).

(f) Recall that b(i) stands for the binary representation of integer i, with no leading zeros. Show that the language of "arithmetic progressions" $L = \{\mathrm{b}(1); \mathrm{b}(2); \ldots; \mathrm{b}(n) : n \geq 1\}$ is not regular. (Use part (e) above.)

(g) Show that L in part (f) above can be decided in space $\log \log n$.

2.8.12 Problem: Show that if a Turing machine uses space that is smaller than $c \log \log n$ for all $c > 0$, then it uses constant space. (This is from

- J. Hartmanis, P. L. Lewis II, R. E. Stearns "Hierarchies of memory-limited computations," *Proc. 6th Annual IEEE Symp. on Switching Circuit Theory and Logic Design*, pp. 179–190, 1965.

Consider a "state" of a machine with input to also include the string contents of the work strings. Then the behavior of the machine on an input prefix can be characterized as a mapping from states to sets of states. Consider now the shortest input that requires space $S > 0$; all its prefixes must exhibit different behaviors—otherwise a shorter input requires space S. But the number of behaviors is doubly exponential in S. Notice that, in view of this result, language L in (g) of Problem 2.8.11 above *requires* $\Theta(\log \log n)$ space.)

2.8.13 Problem: Show that the language of palindromes cannot be decided in less than $\log n$ space. (The argument is a hybrid between Problems 2.8.5 and 2.8.12.)

2.8.14 Problem: Give a detailed proof of Theorem 2.3. That is, give an explicit mathematical construction of the simulating machine in terms of the simulated machine (assume the latter has one string, besides the input string).

2.8.15 Random access machines were introduced in

- J. C. Shepherdson and H. E. Sturgis "Computability of recursive functions," *J.ACM, 10*, 2, pp. 217–255, 1963,

and their performance was analyzed in

- S. A. Cook and R. A. Reckhow "Time-bounded random access machines," *J.CSS, 7*, 4, pp. 354–475, 1973.

2.8.16 Problem: Modify the nondeterministic Turing machine for REACHABILITY in Example 2.10 so that it uses the same amount of space, but always halts.

It turns out that all space-bounded machines, even those that use so very little space that they cannot count steps, can be modified so that they always halt; for the tricky construction see

- M. Sipser "Halting space-bounded computations," *Theor. Comp. Science 10,* pp. 335–338, 1980.

2.8.17 Problem: Show that any language decided by a k-string nondeterministic Turing machine within time $f(n)$ can be decided by a 2-string nondeterministic Turing machine *also* within time $f(n)$. (Discovering the simple solution is an excellent exercise for understanding nondeterminism. Compare with Theorem 2.1 and Problem 2.8.9 for deterministic machines.)

Needless to say, any k-string nondeterministic Turing machine can be simulated by a single-string nondeterministic machine with a quadratic loss of efficiency, exactly as with deterministic machines.

2.8.18 Problem: (a) Show that a nondeterministic one-way finite automaton (recall 2.8.11) can be simulated by a deterministic one, with possibly exponentially many states. (Construct a deterministic automaton each state of which is a *set* of states of the given nondeterministic one.)

(b) Show that the exponential growth in (a) is inherent. (Consider the language $L = \{x\sigma : x \in (\Sigma - \{\sigma\})^*\}$, where the alphabet Σ has n symbols.)

CHAPTER 3

UNDECIDABILITY

Turing machines are so powerful, that they can compute nontrivial facts about Turing machines. Naturally enough, this new power contains the seeds of their ultimate limitations.

3.1 A UNIVERSAL TURING MACHINE

There is one aspect of the models of computation introduced in the previous chapter that makes them appear weaker than actual computers: A computer, appropriately programmed, can solve a wide variety of problems. In contrast, Turing machines and RAM programs seem to specialize in solving a single problem. In this section we point out the rather obvious fact that Turing machines too can be *universal*. In particular, we shall describe a *universal Turing machine* U which, when presented by an input, it interprets this input as the description of another Turing machine, call it M, concatenated with the description of an input to that machine, say x. U's function is to simulate the behavior of M presented with input x. We can write $U(M;x) = M(x)$.

There are some details about U that we must fix first. Since U must simulate an arbitrary Turing machine M, there is no *a priori* bound on the number of states and symbols that U must be prepared to face. We shall therefore assume that the states of all Turing machines are integers, and so are their symbols. In particular, we assume that for any Turing machine $M = (K, \Sigma, \delta, s)$, $\Sigma = \{1, 2, \ldots, |\Sigma|\}$, and $K = \{|\Sigma| + 1, |\Sigma| + 2, \ldots, |\Sigma| + |K|\}$. The starting state s is always $|\Sigma| + 1$. Numbers $|K| + |\Sigma| + 1, \ldots, |K| + |\Sigma| + 6$ will encode the special symbols $\leftarrow$, $\rightarrow$, $-$, h, "yes" and "no". All numbers will be represented for processing by U as binary numbers with $\lceil \log(|K| + |\Sigma| + 6) \rceil$

bits, each with enough leading zeros to make them all of the same length. A description of a Turing machine $M = (K, \Sigma, \delta, s)$ will start with the number $|K|$ in binary, followed by $|\Sigma|$ (these two numbers suffice, by our convention, for defining K and Σ), followed by a description of δ. The description of δ will be a sequence of pairs of the form $((q, \sigma), (p, \rho, D))$, using the symbols "(", ")", ",", etc., all of them assumed to be in the alphabet of U.

We shall use the symbol M to denote this description of machine M. This will never result in notational confusion, as the context will always be clear (recall the clarity and elegant succinctness of the equation $U(M; x) = M(x)$ above).

The description of M on the input of U is followed by a ";", and then a description of x comes. x's symbols are again encoded as binary integers, separated by ",". We shall denote such an encoding of string x also as x.

The universal Turing machine U on input $M; x$ simulates M on input x. The simulation is best described by assuming that U has two strings; we know from Theorem 2.1 that we can effect the same behavior by a single-string machine. U uses its second string to store the current configuration of M. Configuration (w, q, u) is stored precisely as w, q, u, that is, with the encoding of string w followed by a "," which is followed by the encoding of state q, and then by another "," and by the encoding of u. The second string initially contains the encoding of $\triangleright$ followed by the encoding of s and followed by the encoding of x.

To simulate a step of M, U simply scans its second string, until it finds the binary description of an integer corresponding to a state (less than or equal to $|K|$). It searches the first string for a rule of δ matching the current state. If such a rule is located, M moves to the left in the second string to compare the symbol scanned with that of the rule. If there is no match, another rule is sought. If there is a match, the rule is implemented (which entails changing the current symbol and the current state in the second string, and a leftward or rightward motion of the state; the fact that all state and symbol numbers are encoded as binary integers with the same number of bits comes handy here.) When M halts, so does U. This completes our description of the operation of U on $M; x$.

There are many ways for U to discover during its computation that its input fails to encode a Turing machine and its input (you can easily think of at least a dozen). Let us say that, if this happens, U enters a state that moves to the right for ever. Since we shall be interested in the behavior of U only on legitimate inputs, this convention is completely inconsequential.

3.2 THE HALTING PROBLEM

The existence of universal Turing machines leads quickly and inescapably to *undecidable problems;* that is, problems that have no algorithms, languages

that are not recursive. Undecidability has been so much a part of the culture of computer science since its beginnings, that it is easy to forget what a curious fact it is. Strictly speaking, once we accept the identification of problems with languages and algorithms with Turing machines, there are trivial reasons why there must be undecidable problems: There are more languages (uncountably many) than ways of deciding them (Turing machines). Still, that such problems exist so close to our computational ambitions was a complete surprise when it was first observed in the 1930's. Undecidability is in some sense the most lethal form of complexity, and so it deserves early mention in this book. Moreover, the development of the theory of undecidability was an important precursor of complexity theory, and a useful source of methodology, analogies, and paradigms.

We define HALTING to be the following problem: Given the description of a Turing machine M and its input x, will M halt on x? In language form, we let H be the following language over the alphabet of U, the universal machine of the previous section: $H = \{M; x : M(x) \neq \nearrow\}$. That is, language H contains all strings that encode a Turing machine and an input, such that the Turing machine halts on that input. It is easy to establish the following:

Proposition 3.1: H is recursively enumerable.

Proof: We need to design a Turing machine that accepts H, that is, halts with a "yes" output if its input is in H, and never halts otherwise. It is easy to see that the machine needed is essentially our universal Turing machine of the previous section. We only need to modify it slightly so that, whenever it halts, it does so in a "yes" state. □

In fact, H is not just any recursively enumerable language. Among all recursively enumerable languages, HALTING has the following important property: If we had an algorithm for deciding HALTING, then we would be able to derive an algorithm for deciding *any* recursively enumerable language: Let L be an arbitrary recursively enumerable language accepted by Turing machine M. Given input x, we would be able to decide whether $x \in L$ *by simply telling whether* $M; x \in H$. We can say that all recursively enumerable languages can be *reduced* to H. HALTING is then a *complete* problem for the class of recursively enumerable problems. Appropriate concepts of reduction and completeness will be an important tool in our studies of situations in Complexity that are far more involved than this one (which we shall resolve in a moment). Typically, completeness is a sign that the problem cannot be solved satisfactorily.

Indeed, Proposition 3.1 exhausts all the good things we can say about H:

Theorem 3.1: H is not recursive.

Proof: Suppose, for the sake of contradiction, that there is a Turing machine M_H that decides H. Modify M_H to obtain the Turing machine D that behaves as follows: On input M, D first simulates M_H on input $M; M$ until it is about

to halt (it will eventually halt, since it is supposed to decide H). If M_H is about to accept, D enters a state that moves the cursor to the right if its string for ever. If M_H is about to reject, then D halts. Schematically, D operates as follows:

$$D(M): \textbf{if } M_H(M;M) = \text{“yes”} \textbf{ then } \nearrow \textbf{ else “yes”}.$$

What is $D(D)$? That is, *how is D to respond when presented with an encoding of itself?* In particular, does it halt or not?

There can be no satisfactory answer. If $D(D) = \nearrow$, by the definition of D we have that M_H accepts input $D;D$. Since M_H accepts $D;D$, we have that $D;D \in H$, and, by the definition of H, $D(D) \neq \nearrow$.

If on the other hand $D(D) \neq \nearrow$, by D's definition M_H rejects $D;D$, and thus $D;D \notin H$. By the definition of H, his implies $D(D) = \nearrow$.

Since we have arrived at a contradiction by only assuming that M_H decides H, we must conclude that there is no Turing machine deciding H. □

The proof of Theorem 1 employs a familiar argument called *diagonalization*. We are studying a *relation*, namely the set of all pairs (M, x) such that $M(x) \neq \nearrow$. We first notice that the “rows” of the relation (the Turing machines) are also legitimate *columns* (inputs to Turing machines). We then look at the “diagonal” of this huge table (machines with themselves as input) and create a row object (D in our case) that “refutes” these diagonal elements (halts when they don't, doesn't when they do). This leads to immediate contradiction, as D cannot be a row object.

3.3 MORE UNDECIDABILITY

HALTING, our first undecidable problem, spawns a wide range of other undecidable problems. The technique used to this end is, once more, *reduction*. To show that problem A is undecidable we establish that, if there were an algorithm for problem A, then there would be an algorithm for HALTING, which is absurd. We list some examples below:

Proposition 3.2: The following languages are not recursive:
(a) $\{M : M \text{ halts on all inputs}\}$.
(b) $\{M;x : \text{there is a } y \text{ such that } M(x) = y\}$.
(c) $\{M;x : \text{the computation } M \text{ on input } x \text{ uses all states of } M\}$.
(d) $\{M;x;y : M(x) = y\}$.

Proof: (a) We shall reduce HALTING to this problem. Given $M;x$, we construct the following machine M':

$$M'(y): \textbf{if } y = x \textbf{ then } M(x) \textbf{ else halt}.$$

Obviously, M' halts on all inputs if and only if M halts on x. □

The proofs of (b), (c), and (d) above, as well as many other similar undecidability results, are the object of Problem 3.4.1. Also, Theorem 3.2 below is the source of many more undecidability results.

Evidently, recursive languages constitute a proper subset of recursively enumerable ones (recall Propositions 2.1, 3.1 and Theorem 3.1). But there are some simple connections between these two classes that are worth mentioning at this point. First, it is clear that recursive languages do not share the "asymmetry" of recursively enumerable ones.

Proposition 3.3: If L is recursive, then so is $\overline{L}$.

Proof: If M is a Turing machine deciding L, we construct a Turing machine $\overline{M}$ deciding $\overline{L}$ by simply reversing the roles of "yes" and "no" in M. □

In contrast, we now show that $\overline{H}$ is not recursively enumerable, and thus *the class of recursively enumerable languages is not closed under complement.* That is, the asymmetry we have noticed in the definition of acceptance goes very deep (as we shall see in good time, it is an important open problem whether the same is true of nondeterminism). This is a consequence of the following simple fact:

Proposition 3.4: L is recursive if and only if both L and $\overline{L}$ are recursively enumerable.

Proof: Suppose that L is recursive. Then so is $\overline{L}$ (Proposition 3.3), and therefore they are both recursively enumerable.

Conversely, suppose that both L and $\overline{L}$ are recursively enumerable, accepted by Turing machines M and $\overline{M}$, respectively. Then L is decided by Turing machine M', defined as follows. On input x, M' simulates on two different strings both M and $\overline{M}$ *in an interleaved fashion.* That is, it simulates a step of M on one string, then a step of $\overline{M}$ on the other, then again another step of M on the first, and so on. Since M accepts L, $\overline{M}$ accepts its complement, and x must be in one of the two, it follows that, sooner or later, one of the two machines will halt and accept. If M accepts, then M' halts on state "yes". If $\overline{M}$ accepts, then M' halts on "no". □

The etymology of the term "recursively enumerable" is an interesting aside. Suppose that M is a Turing machine, and define the language

$$E(M) = \{x : (s, \triangleright, \epsilon) \overset{M^*}{\rightarrow} (q, y \sqcup x \sqcup, \epsilon) \text{ for some } q, y\}.$$

That is, $E(M)$ is the set of all strings x such that, during M's operation on empty input, there is a time at which M's string ends with $\sqcup x \sqcup$. We say that $E(M)$ is the language *enumerated* by M.

Proposition 3.5: L is recursively enumerable if and only if there is a machine M such that $L = E(M)$.

Proof: Suppose that $L = E(M)$; then L can be accepted by a machine M' which, on input x, simulates M on the empty string, and halts accepting only if during the simulation the string of M ever ends with $\sqcup x \sqcup$. If M ever halts without producing the string $\sqcup x \sqcup$, M' diverges.

The other direction is a little more interesting, since it involves a technique called "dovetailing" (interleaving of computations) that is quite useful; see for example the proof of Theorem 14.1. Suppose that L is indeed recursively enumerable, accepted by machine M. We shall describe a machine M' such that $L = E(M)$. M' on empty input operates as follows: It repeats the following for $i = 1, 2, 3, \ldots$: It simulates M on the i (lexicographically) first inputs, one after the other, *and each for i steps.* If at any point M would halt with "yes" on one of these i inputs, say on x, then M' writes $\sqcup x \sqcup$ at the end of its string before continuing. Under no other circumstances does M' write a $\sqcup$ on its string; this assures that, if $x \in E(M')$, then $x \in L$.

Conversely, suppose that $x \in L$. Then the computation of M on input x halts with "yes" after some number of steps, say t_x steps. Also, suppose that x is the ℓ_xth lexicographically smallest input of M. Then, M' on empty input will eventually set $i := \max\{t_x, \ell_x\}$, will complete its simulation of M on x, and write $\sqcup x \sqcup$ on its string. It follows that $x \in E(M')$.□

We shall next prove a very general result stating essentially that *any non-trivial property of Turing machines is undecidable.* Suppose that M is a Turing machine accepting lanuage L; we write $L = L(M)$. Naturally, not all machines accept languages (for example, a machine may halt with a "no", which is incompatible with our definition of acceptance). If for a machine M there is a string x such that $M(x)$ is neither "yes" nor $\nearrow$, then, by convention, $L(M) = \emptyset$.

Theorem 3.2 (Rice's Theorem): Suppose that $\mathcal{C}$ is a proper, non-empty subset of the set of all recursively enumerable languages. Then the following problem is undecidable: Given a Turing machine M, is $L(M) \in \mathcal{C}$?

Proof: We can assume that $\emptyset \notin \mathcal{C}$ (otherwise, repeat the rest of the argument for the class of all recursively enumerable languages *not* in $\mathcal{C}$, also a proper, nonempty subset of the recursively enumerable languages). Next, since $\mathcal{C}$ is nonempty, we can assume that there is a language $L \in \mathcal{C}$, accepted by machine M_L.

Consider the Turing machine M_H that accepts the undecidable language H (Proposition 3.1), and suppose that we wish to decide whether M_H accepts input x (that is, to solve the HALTING problem for the arbitrary input x). To accomplish this, we construct a machine M_x, whose language is either L or $\emptyset$. On input y, M_x simulates M_H on x. If $M_H(x) =$ "yes", then M_x, instead of halting, goes on to simulate M_L on input y: It either halts and accepts, or never halts, depending on the behavior of M_L on y. Naturally, if $M_H(x) = \nearrow$, $M_x(y) = \nearrow$ as well. Schematically, M_x is this machine:

$$M_x(y) : \textbf{if } M_H(x) = \text{“yes”} \textbf{ then } M_L(y) \textbf{ else } \nearrow .$$

Claim: $L(M_x) \in \mathcal{C}$ if and only if $x \in H$.

In other words, the construction of M_x from x is a *reduction* of HALTING to the problem of telling, given M, whether $L(M) \in \mathcal{C}$. Thus, this latter problem must be undecidable, and the theorem is proved.

Proof of the claim: Suppose that $x \in H$, that is, $M_H(x) =$ “yes”. Then M_x on input y determines this, and then always goes on to accept y, or never halt, depending on whether $y \in L$. Hence the language accepted by M_x is L, which is in $\mathcal{C}$ by assumption.

Suppose then that $M_H(x) = \nearrow$ (since M_H is a machine that accepts language H, this is the only other possibility). In this case M_x never halts, and thus M_x accepts the language $\emptyset$, known not to be in $\mathcal{C}$. The proofs of the claim and the theorem are complete. □

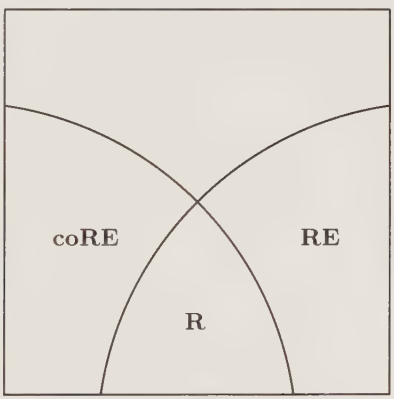


Figure 3-1. Classes of languages.

Recursive Inseparability

We can depict the classes of languages **RE** (the recursively enumerable ones), **coRE** (the complements of recursively enumerable languages) and **R** (the recursive languages) as shown in Figure 3.1. We have already seen representatives of most of the regions shown in the figure. There certainly exist recursive languages, and H is a language that is recursively enumerable but not recursive. Also, it is easy to see that $\overline{H}$, the complement of H, belongs to the region **coRE**−**R**. However, we have not yet seen a representative of the upper region

of the figure. That is, we have not yet encountered a language which is neither a recursively enumerable language, nor the complement of one (the same cardinality argument used in the beginning of this section to establish that not all languages are recursive shows that the *vast majority of languages* are of this type). In Chapter 6 we shall prove that an important language related to logic belongs to this region. In our proof we shall need another kind of "severe undecidability," introduced next.

Consider the "halting" language H and its complement $\overline{H}$. They are examples of what we call *recursively inseparable languages*. Two disjoint languages L_1 and L_2 are called recursively inseparable if there is no recursive language R such that $L_1 \subset R$ and $L_2 \cap R = \emptyset$ (in other words, no recursive R as shown in Figure 3.2 exists). Naturally, this statement is trivial in the case of H and $\overline{H}$, since H and $\overline{H}$ themselves are the only possible separating languages, and we know they are not recursive. However, there are important pairs of disjoint languages that have uncountably many separating languages, none of which is recursive. We give a useful example next.

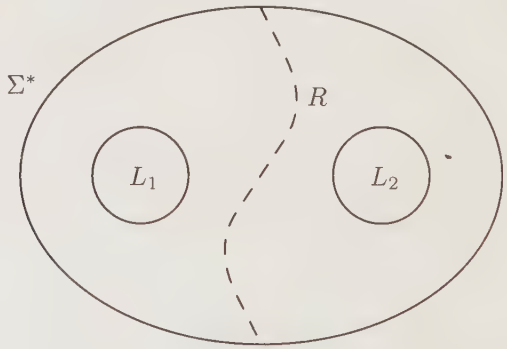


Figure 3-2. Recursively separable languages.

Theorem 3.3: Define $L_1 = \{M : M(M) = \text{"yes"}\}$, and $L_2 = \{M : M(M) = \text{"no"}\}$. Then L_1 and L_2 are recursively inseparable.

Proof: Suppose that there is a recursive language R separating them, that is, $L_2 \subset R$ and $R \cap L_1 = \emptyset$ (if $L_1 \subset R$ and $R \cap L_2 = \emptyset$, then just continue the proof with the complement of R). Consider now the Turing machine M_R deciding R. What is $M_R(M_R)$? It is either "yes" or "no", because M_R decides a language. If it is "yes", then this means that $M_R \in L_1$ (by the definition of L_1), and thus $M_R \notin R$ (because $R \cap L_1 = \emptyset$), and therefore $M_R(M_R) =$ "no" (by the definition of M_R). Similarly, $M_R(M_R) =$ "no" implies $M_R(M_R) =$ "yes", and this is absurd. □

Corollary: Let $L_1' = \{M : M(\epsilon) = \text{“yes”}\}$ and $L_2' = \{M : M(\epsilon) = \text{“no”}\}$. Then L_1' and L_2' are recursively inseparable.

Proof: For any Turing machine M, let M' be the machine which, on any input, generates M's description, and then simulates M on it. We claim that, if we could separate L_1' from L_2' by the recursive language R', then we would be able to recursively separate L_1 and L_2 of the Theorem. This is done as follows: On input M, construct the description of M', and test whether $M' \in R'$. If it is, accept, otherwise, reject. (If input M is not a Turing machine description, then do something arbitrary, say, accept). It is easy to see that the recursive language R decided by this algorithm separates L_1 from L_2. □

Undecidability was first developed in the 1930's with a specific application in mind: Establishing the impossibility of mechanical mathematical reasoning. Mathematical logic is thus our next topic of discussion. In proving our main undecidability result for logic we shall need, three chapters later, the recursive inseparability of L_1' and L_2' in the corollary above.

3.4 NOTES, REFERENCES, AND PROBLEMS

3.4.1 Problem: For each of the following problems involving Turing machines determine whether or not it is recursive.

(a) Given a Turing machine M, does it halt on the empty string?
(b) Given a Turing machine M, is there a string for which it halts?
(c) Given a Turing machine M, does it ever write symbol σ?
(d) Given a Turing machine M, does it ever write a symbol different from the symbol currently scanned?
(e) Given a Turing machine M, is $L(M)$ empty? ($L(M)$ denotes here the language accepted by the Turing machine, *not* decided by it.)
(f) Given a Turing machine M, is $L(M)$ finite?
(g) Given two Turing machines M and M', is $L(M) = L(M')$?

Which of the non-recursive languages above are recursively enumerable?

3.4.2 The Chomsky hierarchy. A *grammar* $G = (\Sigma, N, S, R)$ consists of a set of *terminal symbols* Σ, a set of *nonterminal symbols* N, a start symbol $S \in N$, and a finite set of *rules* $R \subseteq (\Sigma \cup N)^* \times (\Sigma \cup N)^*$. R induces a more complex relation $\rightarrow \subseteq (\Sigma \cup N)^* \times (\Sigma \cup N)^*$, as follows: If $x, y \in (\Sigma \cup N)^*$, we say that $x \rightarrow y$ if and only if $x = x_1 x_2 x_3$, $y = x_1 y_2 x_3$ for some strings x_1, x_2, x_3, y_2, and $(x_2, y_2) \in R$. That is, y is the result of substituting the right-hand side of a rule for an occurrence of the left-hand side of a rule into x. Let $\rightarrow^*$ be *the reflexive-transitive closure* of $\rightarrow$: That is, we define $x \rightarrow^0 y$ if and only if $x = y$, and for $k > 0$ $x \rightarrow^k y$ if and only if for some y' $x \rightarrow^{k-1} y'$ and $y' \rightarrow y$; $x \rightarrow^* y$ if and only if $x \rightarrow^k y$ for some $k \geq 0$. Finally, the *language generated by* G is this: $L(G) = \{x \in \Sigma^* : S \rightarrow^* x\}$. It turns out that grammars and their syntactic restrictions form a most interesting *hierarchy*, first studied by Noam Chomsky

- N. Chomsky "Three models for the description of language," *IRE Transactions on Information Theory, 2*, pp. 113–124, 1956, and
- N. Chomsky "On certain formal properties of grammars," *Information and Control, 2*, pp. 137–167, 1969; for an extensive exposition see
- H. R. Lewis, C. H. Papadimitriou *Elements of the Theory of Computation*, Prentice-Hall, Englewood Cliffs, 1981.

(a) Show that the class of languages generated by grammars is precisely the class of recursively enumerable languages. (In one direction, show how to enumerate all possible "derivations" of strings from S. For the other, show that grammar derivations can be used to simulate the moves of a Turing machine, where the string being manipulated represents the Turing machine's configuration.)

(b) Conclude that it is undecidable, given G and $x \in \Sigma^*$, whether $x \in L(G)$.

Define now a *context-sensitive grammar* to be one for which whenever $(x, y) \in R$ we have $|x| \leq |y|$.

(c) Give a context-sensitive grammar for generating $L = \{xx : x \in \Sigma^*\}$.

(d) Show that it is decidable, given a context-sensitive grammar G and string $x \in \Sigma^*$, whether $x \in L(G)$. (Your algorithm could be nondeterministic.)

In fact, part (d) can be strengthened and made very specific:

(e) Show that the class of languages generated by context-sensitive grammars is precisely **NSPACE**(n). (One direction uses a version of the construction in (a), the other the algorithm in (d).)

A grammar is *context-free* if, for all rules $(x, y) \in R$, $x \in N$; that is, the left-hand side of any rule is a single symbol.

(f) Give context-free grammars that generate the following languages: (i) The palindromes; (ii) strings of "balanced parentheses" (pardon the vagueness, but the only convenient way to define this language is via the context-free grammar sought...); (iii) strings with equal numbers of 0's and 1's.

(g) Show that it is in **P** to determine, given G, whether $\epsilon \in L(G)$.

(h) Show that, if G is a context-free grammar, the problem of determining for any string x whether $x \in L(G)$, is in **P**. (This can be shown by putting G in *Chomsky normal form*, an equivalent grammar with $R \subseteq N \times N^2 \cup N \times \Sigma$, plus possibly the rule (S, ϵ).)

(j) A context-free grammar is *right-linear* if $R \subseteq N \times (\Sigma N \cup \{\epsilon\})$. Show that the class of languages generated by right-linear context-free grammars are precisely the regular languages (the ones decided in constant space, recall Problem 2.8.11).

3.4.3 Post's correspondence problem. In the problem POST we are given a finite list of pairs of strings $P = \{(x_1, y_1), \ldots, (x_n, y_n)\}$, and we are asked whether there is a sequence $(i_1, \ldots, i_m)$ of integers $\leq n$ such that $x_{i_1} x_{i_2} \ldots x_{i_m} = y_{i_1} y_{i_2} \ldots y_{i_m}$. In other words, we are given a dictionary between two languages, and we are asked if any phrase has the same meaning in these languages. Show that POST is undecidable. (Start with the problem in which i_1 is required to be 1.)

3.4.4 Problem: Show that, if L is recursively enumerable, then there is a Turing machine M that enumerates L *without ever repeating an element of* L.

3.4.5 Problem: Show that L is recursive if and only if there is a Turing machine M that enumerates L, and such that the strings in L are output by M *in length-increasing fashion.* (For the *if* direction, if L is finite there is nothing to prove. So suppose it is infinite...)

3.4.6 The "*s-m-n*" Theorem: Show that there is an algorithm which, given a string x and the description of a Turing machine M accepting the language $\{x; y : (x, y) \in R\}$, where R is a relation, constructs the description of a Turing machine M_x accepting the language $\{y : (x, y) \in R\}$.

3.4.7 Problem: Design a Turing machine such that $M(\epsilon)$ is *the description of* M. (Here is a solution, albeit in the informal notation for algorithms used in this book:

> Print the following text twice, the second time in quotes.
> "Print the following text twice, the second time in quotes."

Express this as a Turing machine—or as a program in the programming language of

your choice.)

3.4.8 The recursion theorem: Suppose that f is any recursive function mapping Turing machine descriptions to Turing machine descriptions. Show that there is a Turing machine M such that, for all x, $M(x) = f(M)(x)$. That is, M and $f(M)$ are equivalent, and M is therefore a *fixpoint* of f. (Let g be the recursive function that maps any Turing machine M to another Turing machine, which on any input x computes $M(M)(x)$—if $M(M)$ is not a proper description of a Turing machine, $M(M)(x) = \nearrow$, say. Then there is a Turing machine $M_{f(g)}$ that computes the composition of f with g. Check that the required fixpoint is $g(M_{f(g)})$.)

3.4.9 The arithmetical hierarchy. We say that relation $R \subseteq (\Sigma^*)^k$ is *recursive* if the language $L_R = \{x_1; \ldots; x_k : (x_1, \ldots, x_k) \in R\}$ is recursive. Define Σ_k, for $k \geq 0$, to be the class of all languages L for which there is a recursive $(k+1)$-ary relation R such that

$$L = \{x : \exists x_1 \forall x_2 \ldots Q_k x_k R(x_1, \ldots, x_k, x)\},$$

where Q_k is $\exists$ if k is odd, and $\forall$ if k is even. We define $\Pi_k = \mathbf{co}\Sigma_k$, that is, Π_k is the set of all complements of languages in Σ_k. The languages belonging in Σ_i for some i constitute the *arithmetical hierarchy*.

(a) Show that, for $k \geq 0$, Π_k is the class of all languages L such that there is a recursive relation R such that

$$L = \{x : \forall x_1 \exists x_2 \ldots Q_k x_k R(x_1, \ldots, x_k, x)\}.$$

(b) Show that $\Sigma_0 = \Pi_0 = \mathbf{R}$, and $\Sigma_1 = \mathbf{RE}$.

(c) Show that for all i $\Sigma_{i+1} \supseteq \Pi_i, \Sigma_i$.

(d) Show that Σ_2 is the class of all languages that can be accepted (not decided) by Turing machines that are equipped with the following extra power: At any point the machine may enter a special state $q_?$, and the next state will be $q_{\text{"yes"}}$ or $q_{\text{"no"}}$ depending on whether or not the current contents in its string are the encoding of a halting Turing machine. Extend this definition to Σ_i for $i > 2$. (Compare with oracle machines, studied in Section 14.3.)

(e) Show that for all i $\Sigma_{i+1} \neq \Sigma_i$. (Generalize the proof of the case $i = 0$.)

(f) Show that the language $\{M : L(M) = \Sigma^*\}$ is in $\Pi_2 - \Pi_1$. (To show that it is in Π_2 use (d) above; to show it is not in Π_1, prove that all other languages in Π_2 reduce to it, and use (e).)

(g) Place as low in the arithmetical hierarchy as possible: $\{M : L(M)$ is infinite $\}$.

The arithmetical hierarchy was defined and studied in

- S. C. Kleene "Recursive predicates and quantifiers," *Trans. Amer. Math. Soc., 53*, pp. 41–73, 1943; and
- A. Mostowski "On definable sets of positive integers," *Fundamenta Mathematica, 34*, pp. 81–112.

Compare with the *polynomial hierarchy* defined in Section 17.2.

3.4.10 For much more on the topics of this chapter see

- H. Rogers *Theory of Recursive Functions and Effective Computability*, MIT Press, Cambridge, Massachusetts, 1987 (second edition), and
- M. Davis *Computability and Unsolvability*, McGraw-Hill, New York, 1958.

PART

II LOGIC

Consider the following statement: "Suppose that there is an integer y such that for all integers x we have $x = y + 1$. Then for any two integers w and z, $w = z$." It is a true statement (although trivial, and more than a little confusing and silly). The point is that we can argue about its truth at several levels. This statement can be written as

$$\exists y \forall x (x = y + 1) \Rightarrow \forall w \forall z (w = z).$$

If seen as a statement in number theory, the branch of mathematics that studies the properties of the integers, this sentence is true for a simple reason: Its premise, $\exists y \forall x (x = y + 1)$, is false; clearly there is no number whose successor is every integer.

Thus, the given sentence is of the form $A \Rightarrow B$, where A is a false sentence. It is well-known (and now we are arguing at the level of Boolean logic) that, if A is false, then $A \Rightarrow B$ is true, no matter how horribly wrong B might be. We must conclude that the original sentence is true.

But the original sentence is true for a more fundamental reason, one that has nothing to do with numbers. Let us pretend that we understand nothing about the integers. We know nothing about the meaning of addition, we have no clue what 1 is. We recognize only the meaning of the "logical" symbols in the sentence, such as $\forall$, $=$, $\Rightarrow$, and so on. All we know about $+$ is that it must be a binary function defined in the mysterious universe spoken about in this sentence, and that 1 must be a member of that universe. In this case we read this sentence (and argue about its truth) as a sentence in first-order logic.

As such, the sentence is still a valid one. The argument is this: No matter what $+$ is, surely it is a function, that is, it has a single value for each choice of arguments. If this function is applied to the element y, whose existence is

guaranteed by the premise, and to the element 1*, then we know that we get a particular element of the universe, call it* $y + 1$*. But the premise says that all elements of the universe are equal to* $y + 1$*. That is, any "two" members of the universe must be equal to* $y + 1$*, and thus to each other, that is,* $\forall w \forall z(w = z)$*. The whole sentence is therefore true: The premise indeed implies the conclusion. We have established this result using only the generic properties of functions and of equality.*

These are the three levels of mathematical discourse that will concern us in the next three chapters, in order of sophistication and increasing complexity: Boolean logic, first-order logic, and number theory. Because of the nature of our interest, we shall focus on the many important computational problems related to truth at these three levels. Computational problems from logic are of central importance in complexity theory because, being by nature extremely expressive and versatile (let us not forget that expressibility is logic's raison d' être*), they can be used to express, not just mathematical truth, but computation at various levels. In this book's main project of identifying complexity concepts with actual computational problems, logic will be a most valuable intermediary.*

CHAPTER

4 BOOLEAN LOGIC

The world divides into facts. Each fact can be the case or not the case, while everything else remains the same.

Ludwig Wittgenstein

It can be true, it can be false. You'll be given the same reward.

The Clash

4.1 BOOLEAN EXPRESSIONS

Let us fix a countably infinite alphabet of *Boolean variables* $X = \{x_1, x_2, \ldots\}$. These are variables that can take the two *truth values* **true** and **false**. We can combine Boolean variables using *Boolean connectives* such as $\vee$ (*logical or*), $\wedge$ (*logical and*) and $\neg$ (*logical not*), in much the same way as we combine real variables in algebra by $+$, $\times$, and $-$ to form arithmetic expressions. The definition of Boolean expressions follows.

Definition 4.1: A *Boolean expression* can be any one of (a) a Boolean variable, such as x_i, or (b) an expression of the form $\neg o_1$, where o_1 is a Boolean expression, or (c) an expression of the form $(\phi_1 \vee \phi_2)$, where ϕ_1 and ϕ_2 are Boolean expressions, or (d) an expression of the form $(o_1 \wedge o_2)$ where o_1 and o_2 are Boolean expressions. In case (b) the expression is called *the negation* of ϕ_1; in case (c), it is the *disjunction* of ϕ_1 and ϕ_2; and in case (d), it is the *conjunction* of ϕ_1 and ϕ_2. An expression of the form x_i or $\neg x_i$ is called a *literal*. □

We have defined only the *syntax* of Boolean expressions, that is, their superficial, apparent structure. What gives a logical expression life is its *semantics*, its meaning. The semantics of Boolean expressions is relatively simple: These expressions can be true or false depending on whether the Boolean variables involved are true or false. Now, the definition of a Boolean expression is

inductive, starting from the simplest case of a variable and combining simple expressions by connectives to form more complicated ones. Accordingly, much of our arguing about Boolean expressions will be inductive. Our definitions of properties of Boolean expressions must follow the same inductive path as the original definition, and our proofs will use *induction on the structure of the expression.*

Definition 4.2: A *truth assignment* T is a mapping from a finite set X' of Boolean variables, $X' \subset X$, to the set of *truth values* $\{\textbf{true}, \textbf{false}\}$. Let ϕ be a Boolean expression. We can define the set $X(\phi) \subset X$ of *the Boolean variables appearing in* ϕ inductively as follows: If ϕ is a Boolean variable x_i, then $X(\phi) = \{x_i\}$. If $\phi = \neg\phi_1$, then $X(\phi) = X(\phi_1)$. If $\phi = (\phi_1 \vee \phi_2)$, or if $\phi = (\phi_1 \wedge \phi_2)$, then $X(\phi) = X(\phi_1) \cup X(\phi_2)$.

Now let T be a truth assignment defined on a set X' of Boolean variables such that $X(\phi) \subset X'$; we call such a truth assignment *appropriate to* ϕ. Suppose that T is appropriate to ϕ. We define next what it means for T to *satisfy* ϕ, written $T \models \phi$. If ϕ is a variable $x_i \in X(\phi)$, then $T \models \phi$ if $T(x_i) = \textbf{true}$. If $\phi = \neg\phi_1$, then $T \models \phi$ if $T \not\models \phi_1$ (that is, if it is not the case that $T \models \phi_1$). If $\phi = (\phi_1 \vee \phi_2)$, then $T \models \phi$ if either $T \models \phi_1$ or $T \models \phi_2$. Finally, if $\phi = (\phi_1 \wedge \phi_2)$, $T \models \phi$ if both $T \models \phi_1$ and $T \models \phi_2$ hold. □

Example 4.1: Consider the Boolean expression $\phi = ((\neg x_1 \vee x_2) \wedge x_3)$. An example of an appropriate truth assignment is the one that has $T(x_1) = T(x_3) = \textbf{true}$, and $T(x_2) = \textbf{false}$. Does $T \models \phi$? Clearly $T \not\models x_3$. However, the definition of satisfaction of conjunctions requires that $(\neg x_1 \vee x_2)$ also be satisfied by T. Is it? To begin with, $T \models x_1$, and thus $T \not\models \neg x_1$. Also, $T \not\models x_2$. It follows that $T \not\models (\neg x_1 \vee x_2)$, because it is not the case that one of the disjuncts is satisfied. Finally, $T \not\models \phi$, because it fails to satisfy one of its conjuncts—namely $(\neg x_1 \vee x_2)$. □

We shall find it convenient to use two more Boolean connectives: $(\phi_1 \Rightarrow \phi_2)$ is a shorthand for $(\neg\phi_1 \vee \phi_2)$; and $(\phi_1 \Leftrightarrow \phi_2)$ is a shorthand for $((\phi_1 \Rightarrow \phi_2) \wedge (\phi_2 \Rightarrow \phi_1))$.

We say that two expressions ϕ_1, ϕ_2 are *equivalent*, written $\phi_1 \equiv \phi_2$ if for any truth assignment T appropriate to both of them, $T \models \phi_1$ if and only if $T \models \phi_2$ (this is the same as saying that, for any appropriate T, $T \models (\phi_1 \Leftrightarrow \phi_2)$, see Problem 4.4.2). If two Boolean expressions are equivalent then they can be considered as different representations of one and the same object, and can be used interchangeably. Boolean connectives possess some basic useful properties, such as commutativity and associativity, very much like arithmetic operations. We summarize them next.

Proposition 4.1: Let ϕ_1, ϕ_2, and ϕ_3 be arbitrary Boolean expressions. Then:
(1) $(\phi_1 \vee \phi_2) \equiv (\phi_2 \vee \phi_1)$.
(2) $(\phi_1 \wedge \phi_2) \equiv (\phi_2 \wedge \phi_1)$.

(3) $\neg\neg\phi_1 \equiv \phi_1$.
(4) $((\phi_1 \vee \phi_2) \vee \phi_3) \equiv (\phi_1 \vee (\phi_2 \vee \phi_3))$.
(5) $((\phi_1 \wedge \phi_2) \wedge \phi_3) \equiv (\phi_1 \wedge (\phi_2 \wedge \phi_3))$.
(6) $((\phi_1 \wedge \phi_2) \vee \phi_3) \equiv ((\phi_1 \vee \phi_3) \wedge (\phi_2 \vee \phi_3))$.
(7) $((\phi_1 \vee \phi_2) \wedge \phi_3) \equiv ((\phi_1 \wedge \phi_3) \vee (\phi_2 \wedge \phi_3))$.
(8) $\neg(\phi_1 \vee \phi_2) \equiv (\neg\phi_1 \wedge \neg\phi_3)$.
(9) $\neg(\phi_1 \wedge \phi_2) \equiv (\neg\phi_1 \vee \neg\phi_3)$.
(10) $\phi_1 \vee \phi_1 \equiv \phi_1$.
(11) $\phi_1 \wedge \phi_1 \equiv \phi_1$. □

The commutativity properties (1) and (2) are direct consequences of the symmetry in the definition of $\vee$ and $\wedge$, and property (3) follows directly from the definition of $\neg$. Properties (4) and (5), the associativity properties of $\vee$ and $\wedge$, are also immediate consequences of the definition. For the proofs of properties (6) and (7) (the distributive laws for $\wedge$ and $\vee$), of (8) and (9) (De Morgan's laws), and of (10) and (11) (the idempotency of Boolean expressions) as well as for an introduction to the *truth table method*, a general technique for establishing such facts, see Problem 4.4.2. It is worth noting the complete symmetry, or "duality," between $\vee$ and $\wedge$ apparent in properties (6) through (9). Duality in Boolean expressions is much stronger than in the arithmetic case (where, for example, addition fails to distribute over multiplication).

Proposition 4.1 allows us to use henceforth a simplified notation when representing Boolean expressions. We shall omit parentheses when they separate binary connectives of the same kind ($\vee$ or $\wedge$). That is, we shall write expressions like $(((x_1 \vee \neg x_3) \vee x_2) \vee x_4 \vee (x_2 \vee x_5))$ as $(x_1 \vee \neg x_3 \vee x_2 \vee x_4 \vee x_2 \vee x_5)$, allowing for "long disjunctions" and conjunctions. Notice that, using commutativity and idempotency, we can guarantee that long disjunctions and conjunctions involve distinct expressions: We can rewrite the disjunction above as $(x_1 \vee \neg x_3 \vee x_2 \vee x_4 \vee x_5)$. We shall occasionally use a mathematical notation that parallels the $\sum$ and $\prod$ notation in algebra: $\bigwedge_{i=1}^{n} \phi_i$ stands for $(\phi_1 \wedge \phi_2 \wedge \ldots \wedge \phi_n)$, and similarly $\bigvee_{i=1}^{n} \phi_i$ stands for $(\phi_1 \vee \phi_2 \vee \ldots \vee \phi_n)$.

Furthermore, using these properties we shall establish that every Boolean expression can be rewritten into an equivalent one in a convenient specialized style. In particular, a Boolean expression ϕ is in *conjunctive normal form* if $\phi = \bigwedge_{i=1}^{n} C_i$, where $n \geq 1$, and each of the C_js is the disjunction of one or more literals. The C_js are called the *clauses* of the expression in conjunctive normal form. Symmetrically, we say that an expression ϕ is in *disjunctive normal form* if $\phi = \bigvee_{i=1}^{n} D_i$ where $n \geq 1$, and each of the D_js is the conjunction of one or more literals. The D_js are called the *implicants* of the expression in disjunctive normal form.

Theorem 4.1: Every Boolean expression is equivalent to one in conjunctive normal form, and to one in disjunctive normal form.

Proof: By induction on the structure of ϕ. If $\phi = x_j$, a single variable, the statement is trivially true. If $\phi = \neg\phi_1$, suppose that, by induction, we have turned ϕ_1 into disjunctive normal form with implicants D_j, $j = 1, \ldots, n$. Then, by de Morgan's laws (properties (8) and (9) in Proposition 4.1), ϕ is the conjunction of the $\neg D_j$'s. But each $\neg D_j$ is, by another application of de Morgan's law, the disjunction of literals (the negations of the literals appearing in D_j). Similarly for the disjunctive normal form of ϕ, starting from the conjunctive normal form of ϕ_1.

Suppose now that $\phi = (\phi_1 \vee \phi_2)$. For disjunctive normal form, the induction step is trivial: If both ϕ_1 and ϕ_2 are in disjunctive normal form, then so is $\phi_1 \vee \phi_2$. To put ϕ into conjunctive normal form, suppose that ϕ_1 and ϕ_2 are both already in conjunctive normal form, and let $\{D_{1i} : i = 1, \ldots, n_1\}$ and $\{D_{2j} : j = 1, \ldots, n_2\}$ be the two sets of clauses. Then it is easy to see that the conjunction of the following set of $n_1 \cdot n_2$ clauses is equivalent to ϕ: $\{(D_{1i} \vee D_{2j}) : i = 1, \ldots, n_1, j = 1, \ldots, n_2\}$. The case $\phi = (\phi_1 \wedge \phi_2)$ is completely symmetric. □

The construction in the proof of Theorem 4.1 appears algorithmic enough, and in fact in a promising, polynomial way: The hardest construction in the induction step is the one with the $n_1 \cdot n_2$ clauses, which still can be carried out in time quadratic in the length of the inductively assumed conjunctive normal forms. However, this polynomial appearance is deceiving: Repeated squaring due to a long disjunction can lead to a resulting normal forms that are exponential in the size of the original expressions (see Problem 4.4.5; naturally, exponential is the worst that can happen). In fact, we can assume that our normal forms are a little more standardized: There are no repeated clauses or implicants (if there are, they can be omitted by Proposition 4.1, parts (10) and (11)), and there are no repeated literals in any clause or implicant (same reason).

4.2 SATISFIABILITY AND VALIDITY

We say that a Boolean expression ϕ is *satisfiable* if there is a truth assignment T appropriate to it such that $T \models \phi$. We say that ϕ is *valid* or a *tautology* if $T \models \phi$ for all T appropriate to ϕ. Thus, a valid expression must be satisfiable (and an unsatisfiable expression cannot be valid). If ϕ is valid, we write $\models \phi$, with no mention of a truth assignment T, since any T would do. Furthermore, suppose that a Boolean expression ϕ is unsatisfiable, and consider $\neg\phi$. Obviously, for any truth assignment T appropriate to ϕ, $T \not\models \phi$, and thus $T \models \neg\phi$:

Proposition 4.2: A Boolean expression is unsatisfiable if and only if its negation is valid. □

In other words, the universe of all Boolean expressions is as shown in Figure 4.1, where negation can be thought of as a "flipping" of the figure around its

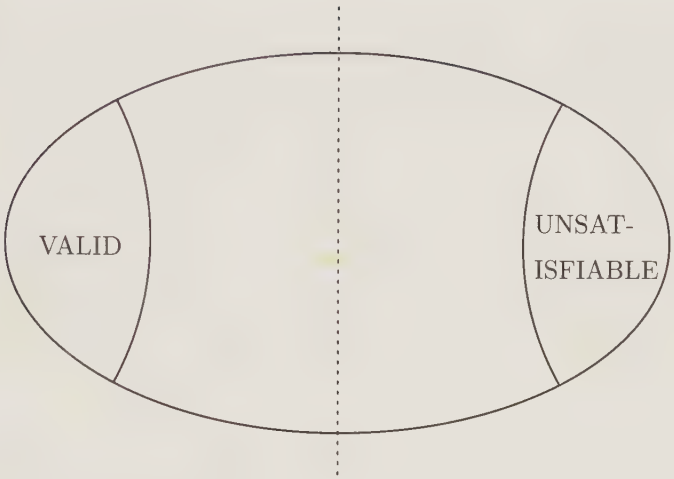


Figure 4-1. The geography of all Boolean expressions.

vertical axis of symmetry.

Example 4.2: Expression $((x_1 \vee \neg x_2) \wedge \neg x_1)$ is satisfiable, satisfied by $T(x_1) = T(x_2) =$ **false**. On the other hand, expression

$$\phi = ((x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3) \wedge (x_3 \vee \neg x_1) \wedge (\neg \vee x_1 \neg x_2 \vee \neg x_3))$$

is not satisfiable. To see this, we first notice that it is in conjunctive normal form, and so a satisfying truth assignment has to satisfy all clauses. The first clause requires that one of the three values $T(x_1), T(x_2), T(x_3)$ is **true**. It is easy to see that the next three clauses require that all three values be the same. Finally, the last clause requires that one of the three values be **false**. It follows that ϕ cannot be satisfied by any truth assignment.

Once we have established this, we also know that $\neg\phi = ((\neg x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (\neg x_1 \wedge x_2) \vee (\neg x_2 \wedge x_3) \vee (\neg x_3 \wedge x_1) \vee (x_1 \wedge x_2 \wedge x_3))$ is valid. □

Satisfiability and validity are important properties of Boolean expressions, so we shall study in great detail the corresponding computational problems. A Boolean expression can be represented for processing by an algorithm as a string over an alphabet that contains the symbols x, 0, and 1 (useful for writing variable names with indices in binary), and also the symbols $(,), \vee, \neg, \wedge$ required by the syntax of Boolean expressions. The *length* of a Boolean expression is the length of the corresponding string. SATISFIABILITY (or SAT, for short) then is the following problem: Given a Boolean expression ϕ in conjunctive normal form, is it satisfiable? We require that the expression be given in conjunctive

normal form for two reasons: First, we know that all expressions can in principle be so represented. And second, this special form of satisfiability seems to capture the intricacy of the whole problem (as Example 4.2 perhaps indicates).

It is of interest to notice immediately that SAT can be solved in $\mathcal{O}(n^2 2^n)$ time by an exhaustive algorithm that tries all possible combinations of truth values for the variables that appear in the expression, and reports "yes" if one of them satisfies it, and "no" otherwise. Besides, SAT can be very easily solved by a *nondeterministic* polynomial algorithm, one that guesses the satisfying truth assignment and checks that it indeed satisfies all clauses; hence SAT is in **NP**. As with another important member of **NP**, TSP (D), presently we do not know whether SAT is in **P** (and we strongly suspect that it is not).

Horn Clauses

Still, there is an interesting *special case* of SAT that can be solved quite easily. We say that a clause is a *Horn clause* if it has *at most one positive literal*. That is, all its literals, except possibly for one, are negations of variables. The following clauses are therefore Horn: $(\neg x_2 \vee x_3)$, $(\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee \neg x_4)$, and (x_1). Of these clauses, the second is a purely negative clause (it has no positive literals), while the rest do have a positive literal, and are called *implications*. They are called so because they can be rewritten as $((x_1 \wedge x_2 \wedge \ldots \wedge x_m) \Rightarrow y)$ where y is the positive literal. For example, the two implications among the three clauses above can be recast as follows: $(x_2 \Rightarrow x_3)$, and and $(\textbf{true} \Rightarrow x_1)$ (in the last clause, the conjunction of no variables was taken to be the "expression" **true**).

Are all these clauses satisfiable? There is an efficient algorithm for testing whether they are, based on the implicational form of Horn clauses. To make the description of the algorithm clear, it is better to consider a truth assignment not as a function from the variables to $\{\textbf{true}, \textbf{false}\}$, but rather as a set T of those variables that are **true**.

We wish to determine whether an expression ϕ, the conjunction of Horn clauses, is satisfiable. Initially, we only consider the implications of ϕ. The algorithm builds a satisfying truth assignment of this part of ϕ. Initially, $T := \emptyset$; that is, all variables are **false**. We then repeat the following step, until all implications are satisfied: Pick any unsatisfied implication $((x_1 \wedge x_2 \wedge \ldots \wedge x_m) \Rightarrow y)$ (that is, a clause in which all the x_is are **true** and y is **false**), and add y to T (make it **true**).

This algorithm will terminate, since T gets bigger at each step. Also, the truth assignment obtained must satisfy all implications in ϕ, since this is the only way for the algorithm to end. Finally, suppose that another truth assignment T' also satisfies all implications in ϕ; we shall show that $T \subseteq T'$. Because, if not, consider the first time during the execution of the algorithm at which T ceased being a subset of T': The clause that caused this insertion to

T cannot be satisfied by T'.

We can now determine the satisfiability of the whole expression ϕ. We claim that *ϕ is satisfiable if and only if the truth assignment T obtained by the algorithm just explained satisfies ϕ.* For suppose that there is a purely negative clause of ϕ that is not satisfied by T—say $(\neg x_1 \vee \neg x_2 \vee \ldots \vee \neg x_m)$. Thus $\{x_1, \ldots, x_m\} \subseteq T$. It follows that no superset of T can satisfy this clause, and we know that all truth assignments that satisfy ϕ are supersets of T.

Since the procedure outlined can obviously be carried out in polynomial time, we have proved the following result (where by HORNSAT we denote the satisfiability problem in the special case of Horn clauses; this is one of many special cases and variants of SAT that we shall encounter in this book).

Theorem 4.2: HORNSAT is in **P**. □

4.3 BOOLEAN FUNCTIONS AND CIRCUITS

Definition 4.3: An *n-ary Boolean function* is a function $f\{\mathbf{true}, \mathbf{false}\}^n \mapsto \{\mathbf{true}, \mathbf{false}\}$. For example, $\vee$, $\wedge$, $\Rightarrow$, and $\Leftrightarrow$ can be thought of as four of the sixteen possible binary Boolean functions, since they map pairs of truth values (those of the constituent Boolean expressions) to $\{\mathbf{true}, \mathbf{false}\}$. $\neg$ is a unary Boolean function (the only other ones are the constant functions and the identity function). More generally, any Boolean expression ϕ can be thought of as an n-ary Boolean function f_ϕ, where $n = |X(\phi)|$, since, for any truth assignment T of the variables involved in ϕ, a truth value of ϕ is defined: **true** if $T \models \phi$, and **false** if $T \not\models \phi$. Formally, we say that Boolean expression ϕ with variables $x_1, \ldots, x_n$ *expresses* the n-ary Boolean function f if, for any n-tuple of truth values $\mathbf{t} = (t_1, \ldots, t_n)$, $f(\mathbf{t})$ is **true** if $T \models \phi$, and $f(\mathbf{t})$ is **false** if $T \not\models \phi$, where $T(x_i) = t_i$ for $i = 1, \ldots, n$. □

So, every Boolean expression expresses some Boolean function. The converse is perhaps a little more interesting.

Proposition 4.3: Any n-ary Boolean function f can be expressed as a Boolean expression ϕ_f involving variables $x_1, \ldots, x_n$.

Proof: Let F be the subset of $\{\mathbf{true}, \mathbf{false}\}^n$ consisting of all n-tuples of truth values such that make f **true**. For each $\mathbf{t} = (t_1, \ldots, t_n) \in F$, let $D_\mathbf{t}$ be the conjunction of all variables x_i with $t_i = \mathbf{true}$, with all negations of variables $\neg x_i$ such that $t_i = \mathbf{false}$. Finally, the required expression is $\phi_f = \bigvee_{\mathbf{t} \in F} D_\mathbf{t}$ (notice that it is already in disjunctive normal form). It is easy to see that, for any truth assignment T appropriate to ϕ, $T \models \phi_f$ if and only if $f(\mathbf{t}) = \mathbf{true}$, where $t_i = T(x_i)$. □

The expression produced in the proof of Proposition 4.3 has length (number of symbols needed to represent it) $\mathcal{O}(n^2 2^n)$. Although many interesting Boolean functions can be represented by quite short expressions, it can be shown that *in*

their great majority, they cannot be (see Theorem 4.3 for a stronger statement).

Definition 4.4: There is a potentially more economical way than expressions for representing Boolean functions —namely *Boolean circuits*. A Boolean circuit is a graph $C = (V, E)$, where the nodes in $V = \{1, \ldots, n\}$ are called the *gates* of C. Graph C has a rather special structure. First, there are no cycles in the graph, so we can assume that all edges are of the form (i, j), where $i < j$ (recall Problem 1.4.4). All nodes in the graph have indegree (number of incoming edges) equal to 0, 1, or 2. Also, each gate $i \in V$ has a *sort* $s(i)$ associated with it, where $s(i) \in \{\textbf{true}, \textbf{false}, \vee, \wedge, \neg\} \cup \{x_1, x_2, \ldots\}$. If $s(i) \in \{\textbf{true}, \textbf{false}\} \cup \{x_1, x_2, \ldots\}$, then the indegree of i is 0, that is, i must have no incoming edges. Gates with no incoming edges are called the *inputs* of C. If $s(i) = \neg$, then i has indegree one. If $s(i) \in \{\vee, \wedge\}$, then the indegree of i must be two. Finally, node n (the largest numbered gate in the circuit, which necessarily has no outgoing edges) is called the *output gate* of the circuit. (In fact, we shall be considering circuits that have *several outputs*, thus computing several functions simultaneously. In this case, any gate with no outgoing edges will be considered an output.)

This concludes our definition of the *syntax* of circuits. The *semantics* of circuits specifies a truth value for each appropriate truth assignment. We let $X(C)$ be the set of all Boolean variables that appear in the circuit C (that is, $X(C) = \{x \in X : s(i) = x \text{ for some gate } i \text{ of } C\}$). We say that a truth assignment T is appropriate for C if it is defined for all variables in $X(C)$. Given such a T, *the truth value of gate* $i \in V$, $T(i)$, is defined, by induction on i, as follows: If $s(i) = \textbf{true}$ then $T(i) = \textbf{true}$, and similarly if $s(i) = \textbf{false}$. If $s(i) \in X$, then $T(i) = T(s(i))$. If now $s(i) = \neg$, then there is a unique gate $j < i$ such that $(j, i) \in E$. By induction, we know $T(j)$, and then $T(i)$ is **true** if $T(j) = \textbf{false}$, and vice versa. If $s(i) = \vee$, then there are two edges (j, i) and (j', i) entering i. $T(i)$ is then **true** if and only if at least one of $T(j)$, $T(j')$ is **true**. If $s(i) = \wedge$, then $T(i)$ is **true** if and only if both $T(j)$ and $T(j')$ are **true**, where (j, i) and (j', i) are the incoming edges. Finally, *the value of the circuit*, $T(C)$, is $T(n)$, where n is the output gate. □

Example 4.3: Figure 4.2(a) shows a circuit. The circuit has no inputs of sort **true** or **false**, and thus it can be thought of as representing a Boolean expression —namely $\phi = ((x_1 \vee (x_1 \wedge x_2)) \vee ((x_1 \wedge x_2) \wedge \neg(x_2 \vee x_3)))$.

Conversely, given a Boolean expression ϕ, there is a simple way to construct a circuit C_ϕ such that, for any T appropriate to both, $T(C_\phi) = \textbf{true}$ if and only if $T \models \phi$. The construction follows the inductive definition of ϕ, and builds a new gate i for each subexpression encountered. The circuit C_ϕ is shown in Figure 4.2(b). Notice the difference between Figures 4.2(a) and 4.2(b). The "standard" circuit C_ϕ is larger, because it does not take advantage of "shared subexpressions." It is the possibility of such shared subexpressions

$$(x_3 \wedge \neg((x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2))) \vee (\neg x_3 \wedge (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2))$$

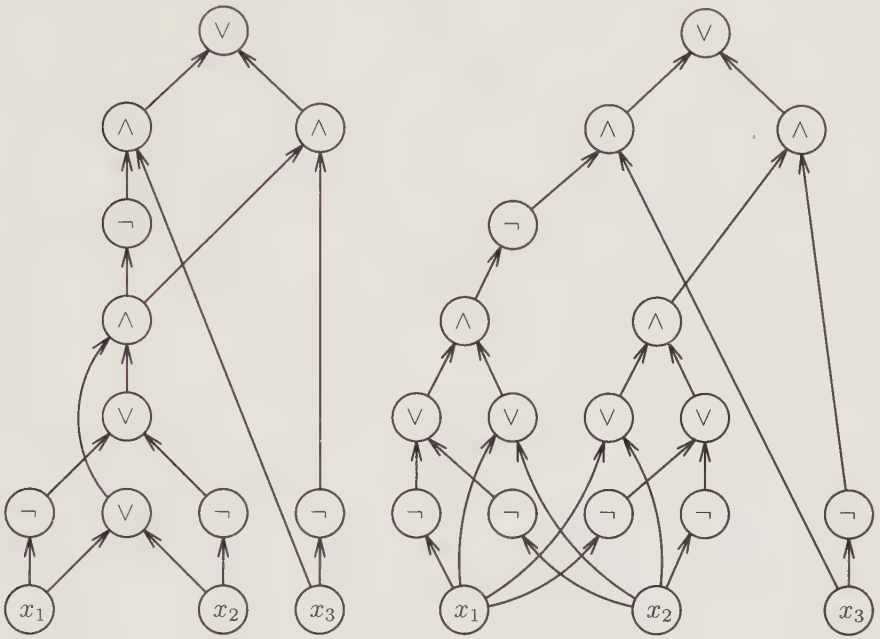


Figure 4-2. Two circuits.

(gates with out-degree greater than one) that make circuits more economical than expressions in representing Boolean functions. □

Example 4.4: Figure 4.3 shows a *variable-free* circuit C, that is, a circuit without gates of sorts in X. Its truth value $T(C)$ is independent of the truth assignment T. In this example, it happens to be **false**. □

There is an interesting computational problem related to circuits, called CIRCUIT SAT. Given a circuit C, is there a truth assignment T appropriate to C such that $T(C) =$ **true**? It is easy to argue that CIRCUIT SAT is computationally equivalent to SAT, and thus presumably very hard; this computational equivalence, or *reduction*, will be shown formally in Chapter 8. Consider, however, the same problem for *circuits with no variable gates* (Figure 4.3). This problem, known as CIRCUIT VALUE, obviously has a polynomial-time algorithm: Compute the values of all gates in numerical order, as in the definition of the

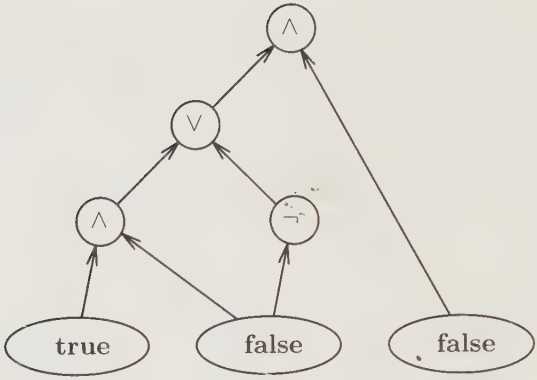


Figure 4-3. A variable-free circuit.

value of a circuit; no truth assignment is required. The CIRCUIT VALUE problem is another one of our fundamental computational problems related to logic.

In analogy with Boolean functions "expressed" by Boolean expressions, Boolean circuits "compute" Boolean functions. For example, both circuits in Figure 4.3 compute the *parity* of x_1, x_2, and x_3 (**true** if an odd number of the variables is **true**, and **false** otherwise). Formally, we say that Boolean circuit C with variables $x_1, \ldots, x_n$ *computes* the n-ary Boolean function f if, for any n-tuple of truth values $\mathbf{t} = (t_1, \ldots, t_n)$, $f(\mathbf{t}) = T(C)$, where the truth assignment T is defined like this: $T(x_i) = t_i$ for $i = 1, \ldots, n$.

Clearly, any n-ary Boolean function can be computed by a Boolean circuit (since it can be expressed as a Boolean expression first). The interesting question is, how large does the circuit need to be, as a function of n?

Theorem 4.3: For any $n \geq 2$ there is an n-ary Boolean function f such that no Boolean circuit with $\frac{2^n}{2n}$ or fewer gates can compute it.

Proof: Suppose, for the sake of contradiction, that for some $n \geq 2$ all n-ary Boolean functions can be computed by circuits with $m = \frac{2^n}{2n}$ or fewer gates.

There are 2^{2^n} different n-ary Boolean functions. But how many circuits with m or fewer gates are there? The precise answer is hard to compute, but it is easy to come up with an overestimate: Each circuit is completely defined once we choose two things for each gate: Its sort, and the gates that feed into it. Thus, for each of the m gates we have at most $(n + 5) \cdot m^2$ choices, or $((n + 5) \cdot m^2)^m$ choices *in toto*. Many of these choices will result in illegal circuits, and several will be guises of the same circuit, but certainly the number of circuits with n variables and m or fewer gates is no larger than $((n+5) \cdot m^2)^m$.

Now it is easy to see that this overestimate on the number of circuits with n variables and m or fewer gates *is less than the number of Boolean functions on n*

variables. To see why, just take the logarithms (in base 2) of the two quantities, remembering that $m = \frac{2^n}{2n}$: The results are $2^n(1 - \frac{\log \frac{4n^2}{n+5}}{2n})$ and 2^n, respectively; and the latter is clearly larger, if we recall that $n \geq 2$. So, having assumed that each Boolean function on n variables is computed by a circuit with at most m gates, we must conclude that *two different n-ary Boolean functions f and f' are computed by the same circuit*, which is of course absurd. □

One of the major frustrations of complexity theory is this: Although we know that many exponentially difficult Boolean functions exist (in fact, most Boolean functions with n inputs are exponentially difficult, see Problem 4.4.14), nobody has been able to come up with a natural family of Boolean functions that requires more than *a linear* number of gates to compute!

4.4 NOTES, REFERENCES, AND PROBLEMS

4.4.1 Most of the material in this chapter, with the possible exception of Theorems 4.2 and 4.3, should be at least indirectly familiar; for a more thorough introduction to Boolean logic the reader is referred to Chapter 8 of

- H. R. Lewis, C. H. Papadimitriou *Elements of the Theory of Computation,* Prentice-Hall, Englewood Cliffs, 1981, and
- H. B. Enderton *A Mathematical Introduction to Logic,* Academic Press, New York, 1972.

4.4.2 Problem: (a) Suppose that two Boolean expressions ϕ_1 and ϕ_2 are such that all truth assignments *appropriate to both* give them the same truth value. Show that this implies that $\phi_1 \equiv \phi_2$.

(b) Use this technique to prove the eleven parts of Proposition 4.1.

This "truth table method" for establishing validity (as well as the term "tautology") was apparently first used in

- L. Wittgenstein *Tractatus Logico-philosophicus*, New York and London, 1922.

The quotation in the beginning of the chapter is from that book, whose main premise was that linguistic utterances about the state of the world can be formalized and studied in terms of logic (see problems 5.8.7 and 5.8.15 for two interesting examples). Wittgenstein in his later work decisively recanted this point of view. The foundations of Boolean logic, of course, go back to George Boole; see

- G. Boole *The Mathematical Analysis of Logic, Being an Essay toward a Calculus for Deductive Reasoning,* Cambridge and London, 1847.

4.4.3 Problem: Show that $\phi_1 \equiv \phi_2$ if and only if $\phi_1 \Leftrightarrow \phi_2$ is a tautology.

4.4.4 Problem: Give the shortest expression you can find equivalent to each of these expressions (**true** and **false** are also allowed as answers, if the expression is either a tautology or unsatisfiable, respectively). Prove the equivalence.

(a) $x \vee \neg x$.
(b) $x \Rightarrow (x \wedge y)$.
(c) $(y \Rightarrow x) \vee x$.
(d) $((x \wedge y) \Leftrightarrow (y \vee z)) \Rightarrow \neg y$.
(e) $\neg((x \wedge y) \Leftrightarrow (x \wedge (y \vee z)))$.
(f) $((x \wedge y) \Rightarrow z) \Leftrightarrow (x \vee (y \wedge z))$.
(g) $((x \Rightarrow y) \Rightarrow (y \Rightarrow z)) \Rightarrow (x \Rightarrow z)$.

4.4.5 Problem: What is the disjunctive normal form of

$$(x_1 \vee y_1) \wedge (x_2 \vee y_2) \wedge \ldots \wedge (x_n \vee y_n)?$$

4.4.6 Theorem 4.2 can be considered as the algorithmic foundation of *logic programming*, an influential style of computer programming; see for example

- R. A. Kowalski "Algorithm = Logic + Control," *C.ACM, 22,* 7, pp. 424–435, 1979.

4.4.7 Problem: We are given a set $\mathcal{T} \subseteq \{\textbf{true}, \textbf{false}\}^n$ of truth assignments. By Proposition 4.3, there is a Boolean expression ϕ such that $\mathcal{T}$ is *precisely* the class of all truth assignments that satisfy ϕ. The question is, under what conditions on $\mathcal{T}$ is ϕ the conjunction of Horn clauses? Call $\mathcal{T}$ *closed under* AND if, whenever $T_1, T_2 \in \mathcal{T}$, the AND of T_1 and T_2 (the truth assignment T with $T(x) = \textbf{true}$ if and only if $T_1(x) = \textbf{true}$ and $T_2(x) = \textbf{true}$) is also in $\mathcal{T}$. Also, we write $T_1 \leq T_2$ if $T_1(x) = \textbf{true}$ implies $T_2(x) = \textbf{true}$.

Show that the following are equivalent:

(i) $\mathcal{T}$ is the set of satisfying truth assignments of some set of Horn clauses.

(ii) $\mathcal{T}$ is closed under AND.

(iii) If $T \in \{\textbf{true}, \textbf{false}\}^n - \mathcal{T}$, then there is at most one T' such that (1) $T' \in \mathcal{T}$; (2) $T \leq T'$; and (3) T' is minimal under $\leq$ with respect to (1) and (2)—that is, if $T'' \in \mathcal{T}$ and $T \leq T''$ then $T' \leq T''$.

(That (i) implies (ii) and (ii) implies (iii) is easy. For the remaining implication, create a Horn clause for each truth assignment not in $\mathcal{T}$.)

4.4.8 Problem: A (rooted) *tree* is a graph $T = (V, E)$ such that there is a unique path from a special node, called the *root*, to every node in V. Each node in a tree is itself a root of a *subtree*, consisting of all nodes to which there is a path from the present node. A tree is called *binary* if each node has at most two outgoing arcs.

Prove *König's lemma:* If a binary tree has infinitely many nodes, then it has an infinitely long path. (Start from the root, and suppose that you have already chosen nodes $1, 2, \ldots, n$ on the path, and furthermore the subtree rooted at n is infinite; show that the next node can be so chosen.)

4.4.9 Problem: Based on König's lemma, prove the *compactness theorem for Boolean logic:* Let S be an infinite set of expressions, and suppose that every finite subset of S is satisfiable. Then S itself is satisfiable. (Let S_i be the expressions in S involving only the first i variables; this set is finite, and thus satisfiable. Define an infinite binary tree, where the nodes at the ith level are the satisfying truth assignments of S_i, and edges signify compatibility. Apply König's lemma.)

4.4.10 Resolution. Suppose that we are given a Boolean expression ϕ in conjunctive normal form, and consider two clauses $C = (x \vee C')$ and $D = (\neg x \vee D')$, where C' is the rest of clause C, and similarly for D and D'. That is, the two clauses contain two opposing literals. Then the clause $(C' \vee D')$, containing all literals of the two clauses except for the two opposing ones is called the *resolvent* of C and D. (If C and D contain only the two opposing literals, then their resolvent is the *empty clause*). Add now the resolvent to ϕ to obtain a new expression ϕ'.

(a) Show that, if ϕ is satisfiable, then so is ϕ'.

We can continue like this, adding to ϕ' a resolvent to obtain ϕ'', and so on. Since we started with a finite number of variables, and thus a finite number of possible clauses, this process must end. Call the result ϕ^*.

(b) Show that ϕ is satisfiable if and only if ϕ^* does not contain the empty clause. (For the proof of the interesting direction, mimic the proof of the compactness theorem.)

The resolution method is an important computational technique for showing unsatisfiability (or validity) in Boolean logic; see

- J. A. Robinson "A machine-oriented logic based on the resolution principle," *Journal of the ACM, 12,* pp. 23-41, 1965.

It is an example of a *sound and complete deduction system*, where (a) above represents soundness, and (b) completeness. For the complexity implications of resolution see Problem 10.4.4.

4.4.11 Problem: There are twenty binary and unary connectives that can appear in Boolean circuits, and our circuits only use three: AND, OR, and NOT. We know that with these three we can express all Boolean functions.

(a) Show that all Boolean functions can be expressed in terms of AND and NOT.

(b) Define now the connectives NOR and NAND as follows: $\text{NOR}(x, y) = \neg(x \vee y)$; and $\text{NAND}(x, y) = \neg(x \wedge y)$. Show that all Boolean functions can be expressed in terms of NOR alone. Repeat for NAND.

(c) Show that there is no other single binary connective besides these two that can express all Boolean functions.

4.4.12 Obviously, circuits are more succinct than expressions (recall Figure 4.2); but how much more? In particular, are there families of Boolean functions for which circuits can be exponentially more succinct than expressions? It turns out that this is a deep question, related to whether *all efficient computation can be parallelized;* see Problem 15.5.4.

4.4.13 Problem: A *monotone* Boolean function F is one that has the following property: If one of the inputs changes from **false** to **true**, the value of the function cannot change from **true** to **false**. Show that F is monotone if and only if it can be expressed as a circuit with only AND and OR gates.

4.4.14 Theorem 4.3 is from

- C. Shannon "The synthesis of two-terminal networks," *Bell System Technical Journal, 28,* pp. 59–98, 1949.

Problem: Extend the proof of Theorem 4.3 to show that the *vast majority* of all Boolean functions with n inputs requires $\Omega(\frac{2^n}{n})$ gates.

CHAPTER 5

FIRST-ORDER LOGIC

First-order logic provides a syntax capable of expressing detailed mathematical statements, semantics that identify a sentence with its intended mathematical application, and a generic proof system that is surprisingly comprehensive.

5.1 THE SYNTAX OF FIRST-ORDER LOGIC

The language we are about to describe is capable of expressing in much more detail than Boolean logic a wide range of mathematical ideas and facts. Its expressions involve constants, functions, and relations from all areas of mathematics. Naturally, every expression will only refer to a small fraction of these. Accordingly, each expression will contain symbols from a particular limited *vocabulary* of functions, constants, and relations.

Definition 5.1: Formally, a *vocabulary* $\Sigma = (\Phi, \Pi, r)$ consists of two disjoint countable sets: A set Φ of *function symbols*, and a set Π of *relation symbols*. r is the *arity* function, mapping $\Phi \cup \Pi$ to the nonnegative integers. Intuitively, r tells us how many arguments each function and relation symbol takes. A function symbol $f \in \Phi$ with $r(f) = k$ is called a *k-ary function symbol*, and a relation symbol $R \in \Pi$ with $r(R) = k$ is also *k-ary*. A 0-ary function symbol is called a *constant*. Relation symbols are never 0-ary. We shall assume that Π always contains the binary *equality relation* $=$. There is also a fixed, countable set of *variables* $V = \{x, y, z, \ldots\}$ which, intuitively, will take values from the universe discussed by the particular expression (and are not to be confused with the Boolean variables of the previous chapter). □

Definition 5.2: We shall now define inductively the *terms* over the vocabulary

Σ. To start off, any variable in V is a term. If $f \in \Phi$ is a k-ary function symbol and $t_1, \ldots, t_k$ are terms, then the expression $f(t_1, \ldots, t_k)$ is a term. Notice that any constant c is a term —just take $k = 0$ in the above definition, and omit the parentheses in $c()$.

Having defined terms, we are finally in a position to start defining what an expression over the vocabulary Σ is. If $R \in \Pi$ is a k-ary relation symbol and $t_1, \ldots, t_k$ are terms, then the expression $R(t_1, \ldots, t_k)$ is called an *atomic expression*. A *first-order expression* is defined inductively as follows: First, any atomic expression is a first-order expression. If ϕ and ψ are expressions, then so are $\neg\phi$, $(\phi \vee \psi)$, and $(\phi \wedge \psi)$. Finally (and here is the most powerful element of this language), if ϕ is a first-order expression, and x is any variable, then $(\forall x\phi)$ is a first-order expression as well. These are all first-order expressions, or *expressions* for short, over Σ. □

We shall use in our expressions the shorthands $\Rightarrow$ and $\Leftrightarrow$ as in Boolean logic (in fact, $\Rightarrow$ will be an important connective in our study of proofs in first-order logic). We shall also use $(\exists x\phi)$ as a shorthand for $\neg(\forall x\neg\phi)$. The symbols $\forall$ and $\exists$ are called *quantifiers*. As we did in Boolean logic, we shall omit parentheses when there is no risk of ambiguity.

A word of caution is in order. You probably realize that something unusual is happening: This is a mathematical book which, at this point, is discussing the language and methodology of mathematics, that is *its own language and methodology*. There is nothing contradictory here, but there is much that is potentially confusing. We have to be very careful in distinguishing between the mathematical language and notation *being studied* (namely, the expressions of first-order logic from various vocabularies), and the mathematical language and notation *employed in this study* (our usual mathematical discourse, as seen in other chapters and other mathematical books, somewhat informal despite extensive use of symbols, yet hopefully rigorous and convincing). This latter language is sometimes referred to as the "metalanguage." To help distinguish between the actual text of a first-order expression (the object of our study) and the various references to it from the metalanguage (the text that records this study), we shall display the text of expressions underlined. In some cases, such as in $(\forall x\phi)$ above, both expression text $(\forall x)$ and metalinguistic symbols (ϕ) coexist in an expression. Such displays will also be underlined.

Example 5.1: *Number theory*, the study of the properties of whole numbers, has fascinated mathematicians since antiquity. We can express sentences about whole numbers in first-order logic by adopting the vocabulary $\Sigma_{\mathbf{N}} = (\Phi_{\mathbf{N}}, \Pi_{\mathbf{N}}, r_{\mathbf{N}})$, defined below. $\Phi_{\mathbf{N}} = \{0, \sigma, +, \times, \uparrow\}$. Of these, 0 is a constant, and σ is a unary function (the successor function). There are three binary functions: $+$ (addition), $\times$ (multiplication), and $\uparrow$ (exponentiation). There is one relation in $\Pi_{\mathbf{N}}$ besides $=$, namely $<$, which is binary.

Here is an expression in number theory:

$$\forall x < (+(x, \sigma(\sigma(0))), \sigma(\uparrow (x, \sigma(\sigma(0))))).$$

It will be convenient, specifically for this vocabulary, to use some special notational simplifications that will make expressions like the one above less alien. First, we shall use the binary functions and predicates in *infix form*, as opposed to prefix; that is, we shall write $(x \times 0)$ instead of $\times(x, 0)$, and $(y < y)$ instead of $< (y, y)$. Also, we shall write $\underline{2}$ instead of the more tedious $\sigma(\sigma(0))$. We do this for any fixed number of applications of σ to $\underline{0}$: $\sigma(\sigma(\sigma(\ldots(\sigma(0))\ldots))$, with 314159 applications of σ, will be written simply $\underline{314159}$. The expression displayed above will be written thus:

$$\forall x((x + 2) < \sigma((x \uparrow 2))).$$

Another expression in number theory is $((2 \times 3) + 3) = ((2 \uparrow 3) + 1)$.

It is probably futile to remind the reader that such expressions are still *just strings*, that we should not rush to assign to them any mathematical meaning. □

Example 5.2: The vocabulary of *graph theory*, $\Sigma_G = (\Phi_G, \Pi_G, r_G)$ is much poorer in comparison. Its scope is to express properties of graphs. It has no function symbols ($\Phi_G = \emptyset$), and it has a single binary relation besides $=$, called G. Typical expressions in graph theory are these: $G(x, x)$; $\exists x(\forall y G(y, x))$; $\forall x(\forall y(G(x, y) \Rightarrow G(y, x)))$; and $\forall x(\forall y(\exists z(G(x, z) \wedge G(z, y)) \Rightarrow G(x, y)))$. □

A variable can appear several times within the text of an expression. For example, $(\forall x(x + y > 0)) \wedge (x > 0)$ contains three such appearances of x. Of these we can disregard the x immediately after $\forall$, because we think of it not really as an appearance of x, but as a part of the "package" $\forall x$. An appearance of a variable in the text of an expression ϕ that does not directly follow a quantifier is called an *occurrence* of x in ϕ. Occurrences can be *free* or *bound*. Intuitively, the first occurrence of x in the expression above, in $x + y > 0$, is not free, because it is referred to by the quantifier $\forall x$. But the second one, in $x > 0$, is free. Formally, if $\forall x \phi$ is an expression, any occurrence of x in ϕ is called *bound*. It is also bound when viewed as an occurrence of x in any expression that contains $\forall x \phi$ as a subexpression. All occurrences that are not bound are *free*. A variable that has a free occurrence in ϕ is a *free variable* of ϕ (even though may have other, bound occurrences). A *sentence* is an expression without free variables.

For example, in $\forall x(x + y > 0) \wedge (x = 0)$, we have two occurrences of x. The first is bound, and the second is free. There is one free occurrence of y. Naturally, this expression is not a sentence, as it has two free variables, x and y. $\forall x(\forall y(\forall z(G(x, z) \wedge G(z, y)) \Rightarrow G(x, y)))$ is a sentence.

5.2 MODELS

The truth of an expression is determined by the values of its constituents, much as in Boolean logic. However, variables, functions, and relations can now take on much more complex values than just **true** and **false**. The analog of a truth assignment for first-order logic is a far more complicated mathematical object called a *model*.

Definition 5.3: Let us fix a vocabulary Σ. A *model appropriate to* Σ is a pair $M = (U, \mu)$. U is a set (*any* non-empty set), called the *universe* of M. μ is a function assigning to each variable, function symbol, and relation symbol in $V \cup \Phi \cup \Pi$ actual objects in U. For each variable x, μ assigns an element $x^M \in U$ (notice that we denote the values of mapping μ by the superscript M, instead of $\mu(\cdot)$). For each k-ary function symbol $f \in \Phi$, μ assigns an actual function $f^M : U \mapsto U^k$ (thus, if $c \in \Phi$ is a constant, c^M is an element of U). Finally, to each k-ary relation symbol $R \in \Pi$, μ assigns an actual relation $R^M \subseteq U^k$. However, to the equality relation symbol $=$, μ is required to assign the relation $=^M$, which is always $\{(u, u) : u \in U\}$.

Suppose now that ϕ is an expression over vocabulary Σ, and that M is a model appropriate to Σ. We shall define when M *satisfies* ϕ, written $M \models \phi$. First, we must define, for an arbitrary term t over Σ, what is its *meaning under* M, t^M. We are already off to a good start: If t is a variable or a constant, t^M is defined explicitly by μ. So, if $t = f(t_1, \ldots, t_k)$, where f is a k-ary function symbol and $t_1, \ldots, t_k$ are terms, then t^M is defined to be $f^M(t_1^M, \ldots, t_k^M)$ (notice that, indeed, it is an element of U). This completes our definition of the semantics of terms.

Suppose then that ϕ is an atomic expression, $\phi = R(t_1, \ldots, t_k)$, where $t_1, \ldots, t_k$ are terms. Then M satisfies ϕ if $(t_1^M, \ldots, t_k^M) \in R^M$ (since we require that $=^M$ is the equality relation on U, there is no reason to treat the case of equality separately). If expression ϕ is not atomic, satisfaction will be defined by induction on the structure of ϕ. If $\phi = \neg\psi$, then $M \models \phi$ if $M \not\models \psi$ (that is, if it is not the case that $M \models \psi$). If $\phi = \psi_1 \vee \psi_2$, $M \models \phi$ if $M \models \psi_1$ or $M \models \psi_2$. If $\phi = \psi_1 \wedge \psi_2$, $M \models \phi$ if $M \models \psi_1$ and $M \models \psi_2$.

So far, the inductive definition parallels that of satisfaction in Boolean logic. The last part is particular to first-order logic: If $\phi = \forall x \psi$, then $M \models \phi$ if the following is true: For any u in U, the universe of M, let $M_{x=u}$ be the model that is identical to M in all details, except that $x^{M_{x=u}} = u$. The requirement is that, for all $u \in U$, $M_{x=u} \models \psi$. □

We shall be mostly studying sentences, that is, expressions with no free variables; our interest in other expressions is essentially limited to the extent that they may be subexpressions of sentences. Now, for sentences, satisfaction by a model does not depend on the values of the variables. This follows from this more general statement:

Proposition 5.1: Suppose that ϕ is an expression, and M and M' are two models appropriate to ϕ's vocabulary such that M and M' agree on everything except for the values they assign to the variables that are *not* free in ϕ. Then $M \models \phi$ if and only if $M' \models \phi$.

Proof: Let us give a careful inductive proof of this statement. Suppose first that ϕ is an atomic expression, and thus all of its variables are free. Then M and M' are identical, and the result is trivial. Suppose then that $\phi = \neg\psi$. The free variables in ϕ are precisely the free variables in ψ, and thus we have: $M \models \phi$ if and only if $M \not\models \psi$ if and only if (by induction) $M' \not\models \psi$ if and only if $M' \models \phi$. Similarly, suppose that $\phi = \psi_1 \wedge \psi_2$. Then the set of free variables in ϕ is the union of the sets of free variables of its constituents. If M and M' disagree only in variables that are not free in ϕ, they do the same with respect to ψ_1 and ψ_2. Thus, we have by induction that $M \models \psi_i$ if and only if $M' \models \psi_i$, $i = 1, 2$. Thus, $M \models \phi$ if and only if $M \models \psi_1$ and $M \models \psi_2$ if and only if (by induction) $M' \models \psi_1$ and $M' \models \psi_2$ if and only if $M' \models \phi$. The case $\phi = \psi_1 \vee \psi_2$ is very similar.

So, let us finally assume that $\phi = \forall x\psi$. The free variables in ϕ are those in ψ, except for x (which may or may not be free in ψ). By the definition of satisfaction, $M \models \phi$ if and only if $M_{x=u} \models \psi$ for all $u \in U$. By the induction hypothesis, the last statement is equivalent to saying that $(M_{x=u})' \models \psi$, for all u, and for any model $(M_{x=u})'$ that disagrees with $M_{x=u}$ only in the non-free variables of ψ. Now, in the last statement we vary the values of the non-free variables of ψ, and the values of x. Thus, we vary all non-free variables of ϕ. We conclude that $M' \models \phi$ for any variant M' of M in the non-free variables of ϕ. □

In view of Proposition 5.1, whether a model satisfies or fails to satisfy an expression does not depend on the values assigned to variables that are bound in the expression (or fail to appear in the expression). By "model appropriate to an expression" we shall henceforth mean the part of a model that deals with the functions, the relations, and the free variables of the expression, if any. We are now turning to some interesting examples of models.

Models of Number Theory

We shall now define a model $\mathbf{N}$, appropriate to the vocabulary of number theory. Its universe is the set of all nonnegative whole numbers. To the constant 0, $\mathbf{N}$ assigns the number $0^{\mathbf{N}} = 0$. To the function σ, $\mathbf{N}$ assigns the unary function $\sigma^{\mathbf{N}}(n) = n + 1$. Similarly, to $+$ it assigns addition, to $\times$ multiplication, and to $\uparrow$ exponentiation. Two numbers m and n are related by $<^{\mathbf{N}}$ if m is less than n. Finally, assume that $\mathbf{N}$ maps all variables to, say, 0 (recall that the values assigned to variables by a model are not very important, Proposition 5.1).

We claim that $\mathbf{N} \models \forall x(x < x + 1)$. To prove this, we have to use our knowl-

edge about the properties of numbers to verify that, for any natural number n, $\mathbf{N}_{x=n} \models \underline{x < x+1}$. That is, we must show that, for all n, $n <^{\mathbf{N}} n +^{\mathbf{N}} 1^{\mathbf{N}}$. But this is equivalent to saying that $n < n+1$ (now not underlined, in the metalanguage), which we know is true of every number n.

On the other hand, $\mathbf{N} \not\models \forall x \exists y(x = y + y)$. The reason is that $\mathbf{N}_{x=1} \not\models \underline{\exists y(x = y + y)}$, or equivalently $\mathbf{N}_{x=1} \models \underline{\forall y \neg(x = y + y)}$, which means that $1 \neq n + n$ for all whole numbers n, which is clearly a true statement.

Arguing about model $\mathbf{N}$ adds another dimension to the slippery distinction between logic and the metalanguage. In order to find out whether $\mathbf{N} \models \phi$, we evoke our mathematical knowledge of the properties of integers, *the very subject to which we are supposed to be gaining some insight* by this exercise. We only do this for the purpose of illustrating the concepts of model and satisfaction. Our ultimate goal is to be able to *mechanize* this process, and develop techniques that discriminate between those sentences that are satisfied by $\mathbf{N}$ (i.e., the *theorems of number theory*) that is and those that are not. In the next chapter, this goal will prove unattainable in a most interesting and devastating way.

Model $\mathbf{N}$ could be called the *standard* model for number theory, because, admittedly, we defined the vocabulary $\Sigma_{\mathbf{N}}$ with this model in mind. However, there are other, *nonstandard models* appropriate for the vocabulary of number theory. For example, the model $\mathbf{N}_p$, where $p > 1$ is an integer, has as universe the set $\{0, 1, \ldots, p-1\}$. $0^{\mathbf{N}_p} = 0$, and all operations are defined *modulo* p. That is, for any m and n in the universe $\sigma^{\mathbf{N}_p}(n) = n \dot{+} 1 \bmod p$, $m +^{\mathbf{N}_p} n = m + n \bmod p$, and similarly for the other operations. For $<$, we say that two whole numbers m and n less than p are related by $<^{\mathbf{N}_p}$ if $m < n$. All variables are mapped to 0, say. Notice that, whether $\mathbf{N}_p \models \underline{\forall x \exists y(x = y + y)}$ depends on the parity of p (if p is odd, $y = \frac{x + p \cdot (x \bmod 2)}{2}$ is the function that establishes that $\mathbf{N}_p \models \underline{\forall x \exists y(x = y + y)}$). However, $\mathbf{N}_p \not\models \underline{\forall x(x < x + 1)}$ (take $x = p - 1$).

For $\mathbf{N}_p$ it is relatively easy to find a sentence that differentiates it from $\mathbf{N}$ (recall the last sentence discussed in the previous paragraph). Unfortunately, there are more "stubborn" nonstandard models of number theory. We shall describe the simplest one, called $\mathbf{N}'$, by defining only σ on it (the other ingredients of number theory can be defined in a compatible manner that we omit). The universe of $\mathbf{N}'$ contains all nonnegative integers, *and all complex numbers of the form* $n+i$, where n is any integer (positive, zero, or negative), and $i = \sqrt{-1}$ is *the imaginary unit*. The successor function according to this nonstandard model maps a nonnegative integer n to $n + 1$, and a complex integer $n + i$ to $(n + 1) + i$. That is, the graph of the successor function, besides the usual half-line of nonnegative integers, contains now a totally disjoint whole line. As we said before, $\mathbf{N}'$ is a very stubborn nonstandard model: We shall show in Section 5.6 that *there is no set of first-order sentences that differentiates between* $\mathbf{N}$ *and* $\mathbf{N}'$*!*

But one can think of models appropriate to $\Sigma_{\mathbf{N}}$ that do not deal with

numbers at all. For example, here is another "model of number theory," called L: The universe of L consists of $2^{\{0,1\}^*}$, the set of all languages over the symbols 0 and 1 (it is an uncountable model). $0^L = \emptyset$, the empty language. For any language ℓ, $\sigma^L(\ell) = \ell^*$. Also, $+^L$ is union, $\times^L$ is concatenation, and $\uparrow^L$ is intersection. Finally, $<^L$ is set inclusion (not necessarily proper). It so happens that $L \models \forall x(x < x + 1)$. In proof, recall that 1 is a shorthand for $\sigma(0)$, and $\sigma^L(0^L) = \emptyset^* = \{\epsilon\}$. And it is true that, for any language x, its union with $\{\epsilon\}$ (as with any other set) contains x as a subset. Therefore, for any language $\ell \subset \{0,1\}^*$, $L_{x=\ell} \models x < x + 1$, and thus $L \models \forall x(x < x + 1)$. It is left to the reader to verify that $L \models \forall x \exists y(x = y + y)$.

Models of Graph Theory

There is an interesting duality between sentences and models. A model may or may not satisfy a sentence. On the other hand, a sentence can be seen as a description of a set of models, namely, those that satisfy it. This point of view is best illustrated in terms of sentences in the universe Σ_G.

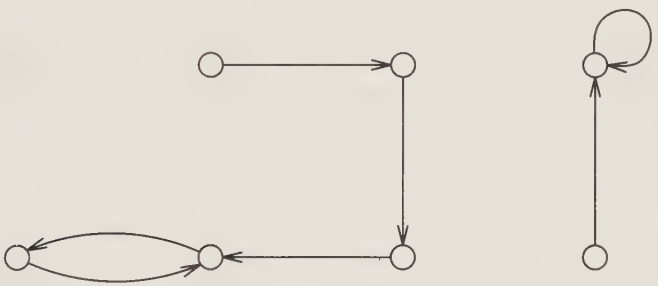


Figure 5-1. A graph that is a function.

Any model appropriate to Σ_G is a graph. We shall be interested only in finite graphs (hence we shall omit "finite" in our discussion). Consider thus the sentence $\phi_1 = (\forall x \exists y G(x,y) \wedge \forall x \forall y \forall z((G(x,y) \wedge G(x,z)) \Rightarrow y = z))$, and the following model Γ for ϕ_1: The universe of Γ is the set of seven nodes of the graph in Figure 5.1, and $G^\Gamma(x,y)$ if there is an edge from x to y in the graph. It should be clear by inspection that $\Gamma \models \phi_1$.

What other graphs satisfy ϕ_1? We claim that all graphs that represent a *function* (that is, all nodes have outdegree one) satisfies ϕ_1, and only these graphs do. In other words, ϕ_1 is a sentence that essentially states "G is a function."

Consider now the sentence $\phi_2 = \forall x(\forall y(G(x,y) \Rightarrow G(y,x)))$. The graph in Figure 5.2 satisfies ϕ_2, but the one in Figure 5.1 does not. ϕ_2 is the statement "G is symmetric." Also, the sentence $\forall x(\forall y(\forall z(G(x,z) \wedge G(z,y)) \Rightarrow G(x,y)))$

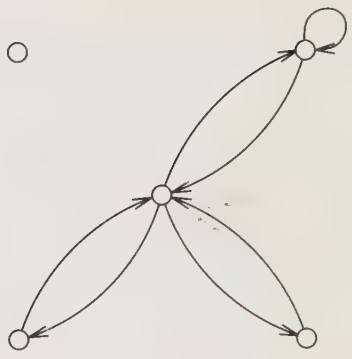


Figure 5-2. A symmetric graph.

states "G is transitive." It is interesting to notice that all these three graph properties (outdegree one, symmetry, and transitivity) can be checked in polynomial time (Problem 5.8.2).

Each sentence in graph theory describes a property of graphs. Now, any property of graphs corresponds in turn to a computational problem: Given a graph G, does it have the property? We are therefore led to the following definition. Let ϕ be any expression over Σ_G (not necessarily a sentence, notice here that we generalize the situation to arbitrary expressions, so we can use induction), define the following problem ϕ-GRAPHS: Given a model Γ for ϕ (that is, a graph G_Γ together with an assignment of nodes of G_Γ to the free variables in ϕ) does $\Gamma \models \phi$? For example, if $\phi = \forall x(\forall y(G(x,y) \Rightarrow G(y,x)))$, ϕ-GRAPHS is SYMMETRY, the computational problem of deciding whether a given graph is symmetric. Such computational problems, involving a graph and certain nodes thereof, are not unknown to us: REACHABILITY from Section 1.1 is such a problem. We next show that any problem of the form ϕ-GRAPHS shares with REACHABILITY an important attribute:

Theorem 5.1: For any expression ϕ over Σ_G, the problem ϕ-GRAPHS is in **P**.

Proof: The proof is by induction on the structure of ϕ. The statement is obviously true when ϕ is an atomic expression of the form $G(x,y)$ or $G(x,x)$. If $\phi = \neg\psi$, then by induction there is a polynomial algorithm that solves the problem ψ-GRAPHS. The same algorithm, with answer reversed from "yes" to "no" and vice-versa, solves ϕ-GRAPHS. If $\phi = \psi_1 \vee \psi_2$, then by induction there are polynomial algorithms that solve ψ_1-GRAPHS and ψ_2-GRAPHS. Our algorithm for ϕ-GRAPHS executes these algorithms one after the other, and replies "yes" if at least one of them answers "yes". The running time of this algorithm is the sum of the two polynomials, and thus a polynomial. Similarly for $\phi = \psi_1 \wedge \psi_2$.

Finally, suppose that $\phi = \forall x\psi$. We know that there is a polynomial-

time algorithm that solves ψ-GRAPHS. Our algorithm for ϕ-GRAPHS repeats the following for each node v of G: Given the alleged model Γ for ϕ, the algorithm attaches the value $x = v$ to Γ to produce a model for ψ (recall that Γ contains no value for x, as x is bound in ϕ). It then tests whether the resulting model satisfies ψ. The algorithm answers "yes" if the answer is "yes" for all vertices v, and answers "no" otherwise. The overall algorithm is polynomial, with a polynomial which equals that for ψ-GRAPHS times n, the number of nodes in the universe of Γ. □

A simple accounting of *space* instead of time in the proof of Theorem 5.1 above gives the following result (which, as we shall see later in Chapter 7, is a stronger statement than Theorem 5.1):

Corollary: For any expression ϕ over Σ_G, the problem ϕ-GRAPHS can be solved in space $\mathcal{O}(\log n)$.

Proof: Problem 5.8.3. □

We shall see later in this chapter that REACHABILITY *cannot* be expressed as ϕ-GRAPHS, for any first-order expression ϕ.

5.3 VALID EXPRESSIONS

For certain expressions we have to struggle to find a model that satisfies them. If such a model exists, we say the expression is *satisfiable*. Some other expressions, however, are satisfied by *any model* (as long as it is appropriate to their vocabulary). Those expressions are called *valid*. If ϕ is valid, we write $\models \phi$, with no reference to models. Intuitively, a valid expression is a statement that is true for very basic reasons, having to do with general properties of functions, quantifiers, equality, etc., and not with the particular mathematical domain (recall the example $(\exists y \forall x (x = y + 1)) \Rightarrow (\forall w \forall z (w = z))$ in the introduction of Part II).

In complete analogy with Proposition 4.2 we have:

Proposition 5.2: An expression is unsatisfiable if and only if its negation is valid. □

Once more, the world of all expressions is as in Figure 5.3.

Boolean Validity

What makes a sentence valid? As it turns out, there are three basic reasons why a first-order expression may be valid. The first one is "inherited" from the world of Boolean expressions. Consider the expression $\phi = \forall x P(x) \vee \neg \forall x P(x)$. ϕ is of the form $\psi \vee \neg \psi$, where $\psi = \forall x P(x)$. Thus it must be a valid sentence, since $\psi \vee \neg \psi$ is a tautology in Boolean logic, when ψ is considered as a Boolean variable. Similarly, the expression $(G(x, y) \wedge G(y, x)) \Rightarrow (G(y, x) \wedge G(x, y))$ can be easily seen to be valid.

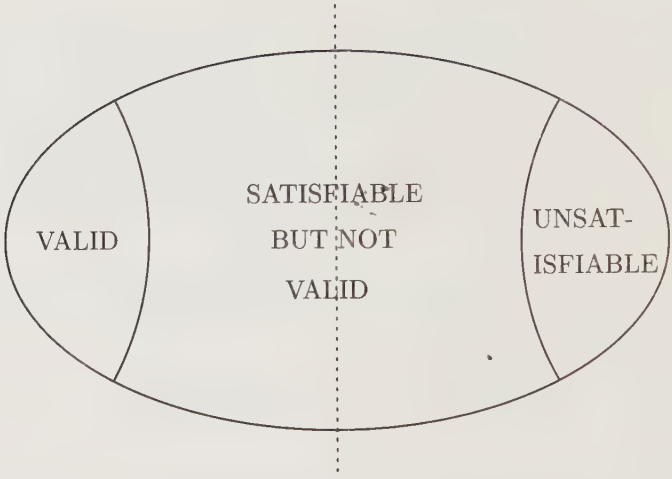


Figure 5-3. First-order expressions (compare with Figure 4.1).

More generally, let ϕ be an expression. We shall define a set of subexpressions of ϕ, called the set of its *principal subexpressions*. Needless to say, the definition will be inductive. Any atomic expression has just one principal subexpression: Itself. Similarly if ϕ is of the form $\forall x\psi$. If now ϕ is of the form $\neg\psi$, then the principal subexpressions of ϕ are precisely those of ψ. Finally, if ϕ is $\psi_1 \vee \psi_2$ or $\psi_1 \wedge \psi_2$, then the set of principal subexpressions of ϕ is the union of the sets of principal subexpressions of ψ_1 and ψ_2.

For example, consider $\forall x G(x,y) \wedge \exists x G(x,y) \wedge (G(z,x) \vee \forall x G(x,y))$. Its principal subexpressions are $\forall x G(x,y)$, $\forall x \neg G(x,y)$, and $G(z,x)$ (notice that we had to first expand the abbreviation $\exists x G(x,y)$ to $\neg\forall x \neg G(x,y)$, before applying the definition). It should be clear that any expression can be thought of as a Boolean expression which has, instead of Boolean variables, the principal subexpressions. We call this the *Boolean form* of the expression. For example, the Boolean form of the above expression is the Boolean expression $x_1 \wedge (\neg x_2) \wedge (x_3 \vee x_1)$, where $x_1 = \forall x G(x,y)$, $x_2 = \forall x \neg G(x,y)$, and $x_3 = G(z,x)$.

Proposition 5.3: Suppose that the Boolean form of an expression ϕ is a tautology. Then ϕ is valid.

Proof: Consider any model M appropriate to the vocabulary of ϕ. For each principal subexpression of ϕ, M either satisfies it or not. This defines a truth assignment to the principal subexpressions, which is guaranteed (by our hypothesis that the Boolean form is a tautology) to satisfy ϕ. □

Boolean logic does not only help identify new valid expressions; it also helps combine expressions already known to be valid to create new valid expressions

(for a systematic study of validity we need rules for both tasks). For example, if we know that ϕ and ψ are valid, we can immediately conclude that $\phi \wedge \psi$ is valid. Also, if ψ and $\psi \Rightarrow \phi$ are valid, then so is ϕ. It turns out that this last rule is the most valuable, as it will be the basis of our proof system in the next section. It is called *modus ponens*, Latin for "way of putting, adding," since it is our proof system's main method for acquiring new valid sentences.

Proposition 5.4 (Modus Ponens): If ψ and $\psi \Rightarrow \phi$ are valid, then ϕ is valid. □

Equality

An expression may also be valid because of the properties of equality. Consider, for example, the expression $x + 1 = x + 1$. It is valid, because in any model $x+1$ will definitely be equal to $x + 1$. (Notice that we are not saying $x + 1 = 1 + x$ is valid; it is *not*.) For a more complex example, $\phi = x = 1 \Rightarrow 1 + 1 = x + 1$ is also valid, independently of the meaning of 1 and +; because, if $x = 1$, then any function applied to arguments $x, 1$ and to $1, 1$ is bound to give the same results (of course, we have used here also the symmetry of equality, that is, the valid expression $t = t' \Rightarrow t' = t$). Similarly, the expression $x = y \Rightarrow (G(x, x) \Rightarrow G(y, x))$ is also valid: If $x = y$, then clearly, $G(x, x)$ implies $G(y, x)$, no matter what G means.

Proposition 5.5: Suppose that $t_1, \ldots, t_k, t'_1, \ldots, t'_k$ are terms. Any expression of the form $t_1 = t_1$, or $(t_1 = t'_1 \wedge, \ldots, \wedge t_k = t'_k) \Rightarrow f(t_1, \ldots, t_k) = f(t'_1, \ldots, t'_k)$, or $(t_1 = t'_1 \wedge, \ldots, \wedge t_k = t'_k) \Rightarrow (R(t_1, \ldots, t_k) \Rightarrow R(t'_1, \ldots, t'_k))$, is valid. □

Quantifiers

So, expressions can be valid for reasons from Boolean logic, and because of the properties of equality. What other reasons for validity are there? The only other important ingredient of first-order logic, besides Boolean connectives and equality, are the *quantifiers*.

It turns out that a sentence may be valid because of the meaning of the quantifiers. For example, consider the expression $G(x, 1) \Rightarrow \exists z G(x, z)$. It is certainly valid. In any model (any graph, that is), if $G(x, 1)$ holds, then clearly there is a z such that $G(x, z)$ holds; namely $z = 1$. Also, the contrapositive reasoning is valid, as exemplified by the expression $\forall x G(x, y) \Rightarrow G(z, y)$. If there is an edge to y from all nodes, then there certainly is one from z, whatever z may be.

To generalize, we need the following notation: Suppose that ϕ is an expression, x a variable, and t a term. We define the *substitution of t for x in ϕ*, denoted $\phi[x \leftarrow t]$, to be the expression obtained by replacing each free occurrence of variable x by the term x. For example, if $\phi = (x = 1) \Rightarrow \exists x (x = y)$

and $t = y + 1$, then $\phi[x \leftarrow t] = (y + 1 = 1) \Rightarrow \exists x(x = y)$, and $\phi[y \leftarrow t] = (x = 1) \Rightarrow \exists x(x = y + 1)$. There is a problem with this definition when t contains a variable which is bound at a position at which x occurs. For example, let us modify slightly the previous expression: If $\phi' = (x = 1) \Rightarrow \exists y(x = y)$, then $\phi'[x \leftarrow t]$ would be $(y + 1 = 1) \Rightarrow \exists y(y + 1 = y)$. In the latter expression, the next to last occurrence of y was "undeservedly" bound by the $\exists y$ quantifier. (To see why this is bad, examine the two expressions, ϕ' and $\phi'[x \leftarrow t]$, no longer pretending that you do not know what addition is.) To avoid such unintended bindings, we say that *t is substitutible for x in ϕ* whenever there is no variable y in t such that some part of ϕ of the form $\forall y\psi$ (or, of course, $\exists y\psi$, which abbreviates $\neg\forall y\neg\psi$) contains a free occurrence of x. *We shall henceforth use the notation $\phi[x \leftarrow t]$ only in cases in which t is substitutible for x in ϕ.* That is, the very usage of this notation contains an implicit assertion that t is substitutible for x in ϕ.

Proposition 5.6: Any expression of the form $\forall x\phi \Rightarrow \phi[x \leftarrow t]$ is valid. □

Notice that by Proposition 5.6, the expression of the form $\phi[x \leftarrow t] \Rightarrow \exists x\phi$ is also valid (it is the contrapositive of $\forall x\phi \Rightarrow \phi[x \leftarrow t]$).

Quantifiers affect validity in another way. Suppose that an expression ϕ is valid. Then we claim that $\forall x\phi$ is also valid. This is because, even if x occurs free in ϕ (the other case being trivial), the fact that ϕ is valid means that it is satisfied by all models, no matter what x is mapped to. But this is the definition of validity for $\forall x\phi$.

Proposition 5.7: If ϕ is valid, then so is $\forall x\phi$. □

There is a stronger form of this. Suppose that x does not occur free in ϕ. Then we claim that $\phi \Rightarrow \forall x\phi$ is valid. The reason is that, any model that satisfies ϕ will also satisfy $\forall x\phi$, as the $x = u$ part of the definition of satisfaction becomes irrelevant. We summarize this as follows:

Proposition 5.8: If x does not appear free in ϕ, then $\phi \Rightarrow \forall x\phi$ is valid. □

Finally, validity also comes from an interesting interaction between quantifiers and Boolean logic: Universal quantifiers distribute over conditionals. That is, $\models ((\forall x(\phi \Rightarrow \psi)) \Rightarrow (\forall x\phi) \Rightarrow (\forall x\psi))$. In proof, the only way for a model M to fail to satisfy this expression is for the following three things to happen: $M \models (\forall x(\phi \Rightarrow \psi))$, $M \models \forall x\phi$, and $M \not\models \forall x\psi$. The third statement says that there is a u for which $M_{x=u} \not\models \psi$. However, we know that $M_{x=u} \models \phi$, and also that $M_{x=u} \models \phi \Rightarrow \psi$. This is a contradiction.

Proposition 5.9: For all ϕ and ψ, $(\forall x(\phi \Rightarrow \psi)) \Rightarrow ((\forall x\phi) \Rightarrow (\forall x\psi))$ is valid. □

Prenex Normal Form

Validity is also a source of convenient simplifications. Clearly, if $\phi \Leftrightarrow \psi$ is valid, then we should feel free to substitute ϕ for ψ. Judicious application of this will

result in simpler expressions (or, in any event, less "chaotic" ones). If $\phi \Leftrightarrow \psi$ is valid, we write $\phi \equiv \psi$. Many such useful equivalences are inherited from Boolean logic (recall Proposition 5.3). The following ones relate quantifiers with Boolean connectives:

Proposition 5.10: Let ϕ and ψ be arbitrary first-order expressions. Then:
(1) $\forall x(\phi \wedge \psi) \equiv (\forall x\phi \wedge \forall x\psi)$.
(2) If x does not appear free in ψ, $\forall x(\phi \wedge \psi) \equiv (\forall x\phi \wedge \psi)$.
(3) If x does not appear free in ψ, $\forall x(\phi \vee \psi) \equiv (\forall x\phi \vee \psi)$.
(4) If y does not appear in ϕ, $\forall x\phi \equiv \forall y\phi[x \leftarrow y]$. □

The first three properties can in fact be proved starting from Propositions 5.7, 5.8, and 5.9. The last property states essentially that the quantified variables of an expression can be given completely new names, so that they do not get in the way of other parts of the expression (for a proof, see Problem 5.8.4).

Using these equivalences, we can prove that any first-order expression can be put into a convenient *normal form*, with all quantifiers in front. In particular, an expression is said to be in *prenex normal form* if it consists of a sequence of quantifiers, followed by an expression that is free of quantifiers (a Boolean combination of atomic expressions).

Example 5.3: Let us turn the following expression into an equivalent one in prenex normal form.

$$(\forall x(G(x,x) \wedge (\forall y G(x,y) \vee \exists y \neg G(y,y))) \wedge G(x,0))$$

To this end, we first apply equivalence (4) in Proposition 5.10 to assign a different variable name to each quantification and each free variable:

$$(\forall x(G(x,x) \wedge (\forall y G(x,y) \vee \exists z \neg G(z,z))) \wedge G(w,0))$$

We then move $\forall x$ outwards, using property (2):

$$\forall x((G(x,x) \wedge (\forall y G(x,y) \vee \exists z \neg G(z,z))) \wedge G(w,0))$$

Next we move $\forall y$ outwards, applying (3) and then (2):

$$\forall x \forall y((G(x,x) \wedge (G(x,y) \vee \exists z \neg G(z,z))) \wedge G(w,0))$$

We can rewrite the innermost parenthesis, using De Morgan's Law (recall Proposition 4.1, Part (8)) to make the $\exists z$ to a $\forall z$.

$$\forall x \forall y((G(x,x) \wedge \neg(\neg G(x,y) \wedge \forall z G(z,z))) \wedge G(w,0))$$

Now $\forall z$ can be moved out:

$$\forall x \forall y((G(x,x) \wedge \neg \forall z(\neg G(x,y) \wedge G(z,z))) \wedge G(w,0))$$

Repeating once more:

$$\underline{\forall x \forall y(\neg(\neg G(x,x) \vee \forall z(\neg G(x,y) \wedge G(z,z))) \wedge G(w,0))}$$

Now $\underline{\forall z}$ can be pulled out:

$$\underline{\forall x \forall y(\neg \forall z(\neg G(x,x) \vee (\neg G(x,y) \wedge G(z,z))) \wedge G(w,0))}$$

Once more De Morgan's Law:

$$\underline{\forall x \forall y \neg(\forall z(\neg G(x,x) \vee (\neg G(x,y) \wedge G(z,z))) \vee \neg G(w,0))}$$

Move $\underline{\forall z}$ out once more:

$$\underline{\forall x \forall y \neg \forall z((\neg G(x,x) \vee (\neg G(x,y) \wedge G(z,z))) \vee \neg G(w,0))}$$

Finally, convert $\underline{\forall z}$ to a $\underline{\exists z}$.

$$\underline{\forall x \forall y \exists z \neg((\neg G(x,x) \vee (\neg G(x,y) \wedge G(z,z))) \vee \neg G(w,0))}$$

We have finally arrived at an expression in prenex normal form. Notice that the expression after the last quantifier (called the *matrix* of the expression in prenex normal form) could be further changed and simplified using the properties of Boolean expressions. For example, it can be put into conjunctive normal form. □

Theorem 5.2: Any first-order expression can be transformed to an equivalent one in prenex normal form.

Proof: See Problem 5.8.5. □

5.4 AXIOMS AND PROOFS

We have equipped our system for expressing mathematical reasoning with syntax and semantics. Surely there is something missing: A systematic method for revealing truth.

But first, what is truth? One possible answer is that truth in first-order logic coincides with the concept of validity. Let us therefore introduce a systematic way for revealing the validity of expressions. Our system is based on the three basic kinds of validity that we know: Boolean validity, the properties of equality, and the properties of quantifiers.

The system we are going to propose works for expressions in any fixed vocabulary Σ. We shall henceforth assume that Σ has been fixed (but in our examples we shall use the usual assortment of familiar vocabularies). Our system starts with a countably infinite (in fact, recursive, see Problem 5.8.6)

AX0: Any expression whose Boolean form is a tautology.

AX1: Any expression of the following forms:

AX1a: $t = t$.

AX1b: $(t_1 = t'_1 \wedge \ldots \wedge t_k = t'_k) \Rightarrow f(t_1, \ldots, t_k) = f(t'_1, \ldots, t'_k)$.

AX1c: $(t_1 = t'_1 \wedge \ldots \wedge t_k = t'_k) \Rightarrow (R(t_1, \ldots, t_k) \Rightarrow R(t'_1, \ldots, t'_k))$.

AX2: Any expression of the form $\forall x \phi \Rightarrow \phi[x \leftarrow t]$.

AX3: Any expression of the form $\phi \Rightarrow \forall x \phi$, with x not free in ϕ.

AX4: Any expression of the form $(\forall x(\phi \Rightarrow \psi)) \Rightarrow (\forall x \phi \Rightarrow \forall x \psi)$.

Figure 5.4. Basic logical axioms.

set of *logical axioms*. The axioms are our basic valid expressions. Our set of logical axioms Λ contains *all expressions of the form discussed in Propositions 5.4, 5.5, 5.6, 5.8, and 5.9,* together with their generalizations, that is, additions of any number of prefixes of the form $\forall x$ (Proposition 5.9). Λ contains all generalizations of the basic axioms displayed in Figure 5.4.

Starting from the axioms, our system generates ("proves") new valid expressions, by a method based on Proposition 5.4. In particular, consider a finite sequence of first-order expressions $S = (\phi_1, \phi_2, \ldots, \phi_n)$, such that, for each expression ϕ_i in the sequence, $1 \leq i \leq n$, either (a) $\phi \in \Lambda$, or (b) there are two expressions of the form ψ, $\psi \Rightarrow \phi$ among the expressions $\phi_1, \ldots, \phi_{i-1}$ (this is *modus ponens*, our way of adding new valid sentences to those we already know, recall the discussion before Proposition 5.4). Then we say that S is a *proof* of expression ϕ_n. The expression ϕ_n is called a *first-order theorem*, and we write $\vdash \phi_n$ (compare with $\models \phi_n$).

Example 5.4: The reflexive property of equality $(x = x)$ is an axiom. The *symmetric property* of equality $x = y \Rightarrow y = x$ is a first-order theorem. Its proof is this:

$\phi_1 = (x = y \wedge x = x) \Rightarrow (x = x \Rightarrow y = x)$ is an axiom from group **AX1c**, where $k = 2$, R is equality, $t_1 = t_2 = t'_2 = x$, and $t'_1 = y$.

$\phi_2 = (x = x)$ is in axiom group **AX1a**.

$\phi_3 = x = x \Rightarrow ((x = y \wedge x = x) \Rightarrow (x = x \Rightarrow y = x)) \Rightarrow$
$(x = y \Rightarrow y = x)$ is in axiom group **AX0** (this one may take a while to check).

$\phi_4 = ((x = y \wedge x = x) \Rightarrow (x = x \Rightarrow y = x)) \Rightarrow (x = y \Rightarrow y = x)$
from ϕ_2 and ϕ_3 by *modus ponens*.

$\phi_5 = (x = y \Rightarrow y = x)$ from ϕ_1 and ϕ_4 by *modus ponens*.
Therefore, $\vdash x = y \Rightarrow y = x$.

The transitivity property of equality, $(x = y \wedge y = z) \Rightarrow x = z$, is also a first-order theorem (the proof is similar, see Problem 5.8.8). □

We shall give more examples of proofs later, after we develop a methodology that will make proofs a little less tedious than this.

But let us not forget our interest in identifying important computational problems. First-order expressions can be encoded as strings in an appropriate alphabet. Let us fix a vocabulary $\Sigma = (\Phi, \Pi, r)$. A possible encoding would use the symbols F, R, x, 0, and 1 (for expressing functions, relations and variables, all with binary indices), along with the logical symbols $\wedge$, $\vee$, $\neg$, $\exists$, and $\forall$, and parentheses. The same alphabet can, of course, encode proofs (which after all are just sequences of expressions). There is an algorithm for checking whether a string is a proof: Examine all expressions one by one, and determine for each whether it belongs to one of the groups of our axioms for first-order logic. This is not trivial, but not very hard (see Problem 5.8.6). Then, for each expression that is not an axiom, test whether it follows from two of the previous expressions by *modus ponens*.

There are some important computational questions one may ask here: Given an (encoding of an) expression ϕ, is it the case that $\vdash \phi$, that is, is ϕ a first-order theorem? We call this problem THEOREMHOOD.

Proposition 5.11: THEOREMHOOD is recursively enumerable.

Proof: The Turing machine that accepts the language of THEOREMHOOD tries all possible proofs (finite sequences of expressions), in lexicographic order, and reports "yes" if one of them is indeed a proof of the given expression. □

Another problem asks: Given ϕ, is it valid? We call this problem VALIDITY. The definition of this problem is computationally fearsome: It appears to require that all possible models of a sentence (of which there are uncountably many!) must be checked. At this point, we have no clue on whether VALIDITY is recursively enumerable. We shall show in the next section that it is, by establishing a surprising fact: VALIDITY *coincides with* THEOREMHOOD; that is, $\models \phi$ if and only if $\vdash \phi$ (a special case of Theorem 5.7 below). In other words, our tedious and clumsy proof system turns out as powerful as it can be! This would be more than enough for our project of systematizing our search for mathematical truth, *if validity in first-order logic were a satisfactory notion of mathematical truth.*

But, of course, it is not. As mathematicians, we are really interested in discovering the properties of the universes spoken about in the various vocabularies: The whole numbers and their operations, the real numbers, the graphs. First-order logic is just a notation that allows us to study reasoning about all these interesting universes in a unified way. At any point we want to know

whether a sentence is satisfied not by *all models* (this would be validity), but by *our favorite model*, whatever this may be at the time.

How can we bridge the gap between what we can do (systematize validity) and what we need (systematize truth in our favorite model)? A very natural way is the *axiomatic method*. To illustrate the method in an unreasonably favorable case, suppose that we are interested in all sentences that are satisfied by a particular model M_0. Let us assume that we have discovered an expression ϕ_0 such that $M_0 \models \phi_0$, and, furthermore the following is true: $M_0 \models \phi$ if and only if $\models \phi_0 \Rightarrow \phi$. Hence, in this case we could use validity in order to study truth in our "favorite model," M_0.

Example 5.5: Axiomatization is not as hopeless a task as it seems. Some interesting parts of mathematics can indeed be axiomatized. For example, group theory has a vocabulary consisting of the binary function $\circ$ (in infix notation) and the constant 1. A typical expression in this vocabulary would be $\forall x(x \circ y) = x$. All properties of groups can be derived from the following non-logical axioms:

GR1: $\forall x \forall y \forall z((x \circ y) \circ z = x \circ (y \circ z))$ (associativity of $\circ$).

GR2: $\forall x(x \circ 1) = x$ (1 is the identity).

GR3: $\forall x \exists y(x \circ y = 1)$ (existence of inverses).

(We could have also included in the vocabulary of group theory a unary function $(\cdot)^{-1}$, standing for the inverse. However, this is not necessary, since we can think that axiom **GR3** *defines the inverse function*. Its uniqueness follows from the other axioms.) These three simple axioms comprise a *complete axiomatization* of groups. All properties of groups can be deduced from these by our proof method. If we wanted to axiomatize *Abelian* groups, we would have added the axiom

GR4: $\forall x \forall y(x \circ y) = (y \circ z)$.

On the other hand, if we wanted to study *infinite* groups, it would suffice to add the sentence $\phi_n = \exists x_1 \exists x_2, \ldots \exists x_n \bigwedge_{i \neq j}(x_i \neq x_j)$, *for each* $n > 1$. This infinite set of sentences is a complete axiomatization of infinite groups. □

In general, however, our favorite model may have an axiomatization that consists of an infinite set of expressions (recall the infinite groups in the previous example). So, our proof system must be generalized to allow for *proofs from infinitely many premises*. Let Δ be a set of expressions, and ϕ another expression. We say that ϕ *is a valid consequence of* Δ, written $\Delta \models \phi$, if any model that satisfies each expression in Δ must also satisfy ϕ. In our intended application to axiomatization, presumably the valid consequences of Δ

will be all properties of M_0, and just these. So, we are very much interested in systematically generating all valid consequences of Δ.

We introduce next a proof system, which is a natural extension of the one above for validity, and is helpful in identifying valid consequences. Let Δ be a set of expressions. Let S be a finite sequence of first-order expressions $S = (\phi_1, \phi_2, \ldots, \phi_n)$, such that, for each expression ϕ_i in the sequence, $1 \leq i \leq n$, either (a) $\phi \in \Lambda$, or (b) $\phi \in \Delta$, or (c) there are two expressions of the form ψ, $\underline{\psi \Rightarrow \phi}$ among the expressions $\phi_1, \ldots, \phi_{i-1}$. Then we say that S is a *proof of* $\underline{\phi = \phi_n}$ *from* Δ. The expression ϕ is then called a Δ-*first-order theorem* , and we write $\Delta \vdash \phi$ (compare with $\Delta \models \phi$).

In other words, a Δ-first-order theorem would be an ordinary first-order theorem *if* we allowed all expressions in Δ to be added to our logical axioms. In this context, the expressions in Δ are called the *nonlogical axioms* of our system†.

Using the concept of proof from a set of expressions we can simplify even proofs of validity (where no premises are needed). This is done by exploiting three interesting results, stated next. These results formalize three patterns of thought very common in mathematical reasoning: they could be called the *deduction technique*, *arguing by contradiction*, and *justified generalization*.

In the deduction technique we wish to prove $\underline{\phi \Rightarrow \psi}$ and we argue thus: "Let us assume that ϕ holds..."

Theorem 5.3 (The Deduction Technique): Suppose that $\Delta \cup \{\phi\} \vdash \psi$. Then $\Delta \vdash \underline{\phi \Rightarrow \psi}$.

Proof: Consider a proof $S = (\phi_1, \ldots, \phi_n)$ of ψ from $\Delta \cup \{\phi\}$ (that is, $\phi_n = \psi$). We shall prove by induction on i that there is a proof of $\underline{\phi \Rightarrow \phi_i}$ from Δ, for $i = 0, \ldots, n$. The result will follow, taking $i = n$.

The statement is vacuously true when $i = 0$, so suppose it is true for all $j < i$, where $i \leq n$. The proof of $\underline{\phi \Rightarrow \phi_i}$ includes all proofs of the expressions $\underline{\phi \Rightarrow \phi_j}$, $1 \leq j < i$, and certain new expressions that prove $\underline{\phi \Rightarrow \phi_i}$, depending on the nature of ϕ_i. If ϕ_i is in $\Delta \cup \Lambda$ (that is, it is a logical or non-logical axiom), we add to the proof the expressions $\ldots, \phi_i, \underline{\phi_i \Rightarrow (\phi \Rightarrow \phi_i)}$, and $\underline{\phi \Rightarrow \phi_i}$. The first expression can be added as an axiom (it is one, by our assumption); the second as an axiom of the Boolean group **AX0**; and the third is added by *modus ponens*. If ϕ_i in S was obtained from some ϕ_j and $\underline{\phi_j \Rightarrow \phi_i}$ by *modus ponens*, with $j < i$, remember that, by induction, our proof now includes $\underline{\phi \Rightarrow \phi_j}$ and $\underline{\phi \Rightarrow (\phi_j \Rightarrow \phi_i)}$. We add to our proof the expressions $\underline{(\phi \Rightarrow \phi_j) \Rightarrow ((\phi \Rightarrow (\phi_j \Rightarrow \phi_i)) \Rightarrow (\phi \Rightarrow \phi_i))}$, $\underline{(\phi \Rightarrow (\phi_j \Rightarrow \phi_i)) \Rightarrow (\phi \Rightarrow \phi_i)}$, and

† The term *non-logical* makes the distinction between the expressions in Δ and the *logical axioms* in Λ. It contains no derogatory implication about the expressions in Δ being somehow "illogical."

$\phi \Rightarrow \phi_i$. The first expression is an axiom of group **AX0**, while the second and the third are added by *modus ponens*.

Finally, if it is the case that $\phi_i = \phi$, then we can add $\phi \Rightarrow \phi$ to our proof as an axiom from group **AX0**. □

The next proof method is perhaps even more familiar: To show ϕ, we assume $\neg\phi$ and arrive at a contradiction. Formally, a *contradiction* can be defined as the expression $\psi \wedge \neg\psi$, where ψ is some arbitrary expression; if $\psi \wedge \neg\psi$ can be proved from Δ, then all expressions, including all other contradictions, can be proved as tautological implications. If $\Delta \vdash \phi$, for *any* expression ϕ (including the contradictions mentioned above), then we say that Δ is *inconsistent*; otherwise, if no contradiction can be proved from Δ, we say Δ is *consistent*.

Theorem 5.4 (Arguing by Contradiction): If $\Delta \cup \{\neg\phi\}$ is inconsistent, then $\Delta \vdash \phi$.

Proof: Suppose that $\Delta \cup \{\neg\phi\}$ is inconsistent. Then $\Delta \cup \{\neg\phi\} \vdash \phi$ (along with any other expression). By Theorem 5.3, we know that $\Delta \vdash \neg\phi \Rightarrow \phi$, which is equivalent to ϕ. Formally, our proof adds to the proof of $\neg\phi \Rightarrow \phi$ the sequence $\ldots (\neg\phi \Rightarrow \phi) \Rightarrow \phi, \phi$, the former as a Boolean axiom, the latter by *modus ponens*. □

The last argument in the proof of Theorem 5.4 (proving ϕ from $\neg\phi \Rightarrow \phi$) is very common, and quite routine. It involves proving an expression which the "Boolean consequence" of already proven expressions. In the future we shall telescope such steps, bringing in Boolean consequences without much ado.

In mathematics we frequently end our proofs as follows: "...and since x was taken to be an arbitrary integer, the proof is complete." This type of argument can be formalized:

Theorem 5.5 (Justified Generalization): Suppose that $\Delta \vdash \phi$, and x is not free in any expression of Δ. Then $\Delta \vdash \forall x\phi$.

Proof: Consider a proof $S = (\phi_1, \ldots, \phi_n)$ of ϕ from Δ (that is, $\phi_n = \phi$). We shall prove by induction on i that there is a proof of $\forall x\phi_i$ from Δ, for $i = 0, \ldots, n$. The result will follow, taking $i = n$.

The statement is vacuously true when $i = 0$, so suppose it is true for all $j < i$, where $i \leq n$. The proof of $\phi \Rightarrow \phi_i$ includes all proofs of the expressions $\forall x\phi_j$, $1 \leq j < i$, and certain new expressions that prove $\forall x\phi_i$. These new expressions depend on ϕ_i. If ϕ_i is in Λ (that is, it is a logical axiom), then $\forall x\phi_i$ is an axiom as well, and can be added to the proof. If ϕ_i is in Δ (that is, it is a non-logical axiom), then we know x is not free in ϕ_i, and thus we add the sequence $\ldots, \phi_i, (\phi_i \Rightarrow \forall x\phi_i), \forall x\phi_i$. The first is a non-logical axiom, the second an axiom from group **AX3**, and the last is added by *modus ponens*. If finally ϕ_i was obtained from some ϕ_j and $\phi_j \Rightarrow \phi_i$ by *modus ponens*, by induction our proof now includes $\forall x\phi_j$ and $\forall x(\phi_j \Rightarrow \phi_i)$. We add to the proof the expressions

$\ldots \forall x(\phi_j \Rightarrow \phi_i) \Rightarrow ((\forall x\phi_j) \Rightarrow (\forall x\phi_i))$, $(\forall x\phi_j) \Rightarrow (\forall x\phi_i)$, and $\forall x\phi_i$). The first is an axiom of group **AX4**, while the second and the third are obtained by *modus ponens*. □

We illustrate these new concepts and proof techniques below:

Example 5.6: It turns out that $\vdash \forall x\forall y\phi \Rightarrow \forall y\forall x\phi$. This is the familiar fact that the order of successive quantifiers *of the same kind* does not matter in logical expressions. Here is its proof:

$\phi_1 = \forall x\forall y\phi$; to use the deduction method, we are assuming the premise of the desired expression. That is, we assume that $\Delta = \{\phi_1\} = \{\forall x\forall y\phi\}$.

$\phi_2 = \forall x\forall y\phi \Rightarrow \forall y\phi$; this is an axiom from group **AX2**.

$\phi_3 = \forall y\phi \Rightarrow \phi$, also an axiom from group **AX2**.

$\phi_4 = \phi$; a Boolean consequence of the previous three expressions.

$\phi_5 = \forall x\phi$. This is an application of justified generalization. Notice that x is not free in any expression in Δ (of which there is only one, $\forall x\forall y\phi$).

$\phi_6 = \forall y\forall x\phi$, again by justified generalization.

Since we have shown that $\{\forall x\forall y\phi\} \vdash \forall y\forall x\phi$, by the deduction technique we conclude that $\vdash \forall x\forall y\phi \Rightarrow \forall y\forall x\phi$. □

Example 5.7: Let us prove that $\vdash \forall x\phi \Rightarrow \exists x\phi$.

$\phi_1 = \forall x\phi$, to apply the deduction technique.

$\phi_2 = (\forall x\phi) \Rightarrow \phi$ is an axiom from group **AX2**.

$\phi_3 = \phi$, by *modus ponens*.

$\phi_4 = \forall x\neg\phi \Rightarrow \neg\phi$ is an axiom from group **AX2**.

$\phi_5 = (\forall x\neg\phi \Rightarrow \neg\phi) \Rightarrow (\phi \Rightarrow \exists x\phi)$, a Boolean axiom, if we recall that $\exists$ is an abbreviation.

$\phi_6 = \phi \Rightarrow \exists x\phi$ by *modus ponens*.

$\phi_7 = \exists x\phi$ by *modus ponens* on ϕ_3 and ϕ_6. □

Example 5.8: We shall prove now the following fact: Suppose that ϕ and ψ are two expressions that are identical, except for the following difference: ϕ has free occurrences of x exactly at those positions that ψ has free occurrences of y. Then $\vdash \forall x\phi \Rightarrow \forall y\psi$. This is the familiar fact stating that *the precise names of variables bound by quantifiers are not important*. Stated otherwise, if an expression is valid, then so are all of its *alphabetical variants*. Here is the proof:

$\phi_1 = \forall x\phi$, we are using the deduction technique.

$\phi_2 = \forall x\phi \Rightarrow \psi$, an axiom in group **AX2**, $\psi = \phi[x \leftarrow y]$.

$\phi_3 = \psi$, by *modus ponens*.

$\phi_4 = \forall y\psi$, by justified generalization; y is not free in $\forall x\phi$. □

We next prove a reassuring result, stating that our proof system is *sound*, in that it only proves valid consequences.

Theorem 5.6 (The Soundness Theorem): If $\Delta \vdash \phi$ then $\Delta \models \phi$.

Proof: Consider any proof $S = (\phi_1, \ldots, \phi_n)$ from Δ. We shall show by induction that $\Delta \models \phi_i$. If ϕ_i is a logical or non-logical axiom, then clearly $\Delta \models \phi_i$. So, suppose that ϕ_i is obtained from ϕ_j, $j < i$, and $\phi_j \Rightarrow \phi_i$ by *modus ponens*. Then, by induction, $\Delta \models \phi_j$ and $\Delta \models \phi_j \Rightarrow \phi_i$. Thus any model that satisfies Δ also satisfies ϕ_j and $\phi_j \Rightarrow \phi_i$; therefore it satisfies ϕ_i. It follows that $\Delta \models \phi_i$. □

5.5 THE COMPLETENESS THEOREM

The converse of the soundness theorem, the completeness theorem for first-order logic due to Kurt Gödel, states that the proof system introduced in the previous section is able to prove all valid consequences:

Theorem 5.7 (Gödel's Completeness Theorem): If $\Delta \models \phi$ then $\Delta \vdash \phi$.

We shall prove a variant of this result, namely the following:

Theorem 5.7 (Gödel's Completeness Theorem, Second Form): If Δ is consistent, then it has a model.

To show that the original form follows from the second, suppose that $\Delta \models \phi$. This means that any model that satisfies all expressions in Δ, also satisfies ϕ (and, naturally, falsifies $\neg\phi$). Hence, no model satisfies all expressions in $\Delta \cup \{\neg\phi\}$, and hence (here we are using the second form of the theorem) this set is inconsistent. Arguing by contradiction (Theorem 5.4), $\Delta \vdash \phi$. Also, the second form can be easily seen to be a consequence of the first (see Problem 5.8.10). We shall next prove the second form.

Proof: We are given a set Δ of expressions over a vocabulary Σ. We know that Δ is consistent. Based on this information alone, we must find a model for Δ. The task seems formidable: We must somehow build a semantic structure from the syntactic information we currently possess. The solution is simple and ingenious: *Our model will be a syntactic one*, based on the elements of our language (there are several other instances in computer science and logic where we resort to semantics by syntax). In particular, *our universe will be the set of all terms over* Σ. And the details of the model (the values of the relations and functions) will be defined as mandated by the expressions in Δ.

There are serious difficulties in implementing this plan. First, the universe of all terms may not be rich enough to provide a model: Consider $\Delta = \{\exists x P(x)\} \cup \{\neg P(t) : t \text{ is a term over } \Sigma\}$. Although this is a consistent set of expressions, it is easy to see that there is no way to satisfy it by a model whose universe contains just the terms of Σ. Another difficulty is that the expressions

in Δ may be two few and inconclusive to drive our definition of the model. For a certain term t, they may provide no clue on whether $P(t)$ or not.

To avoid these difficulties, we employ a maneuver due to Leon Henkin, explained below. First, we add to Σ a countable infinity of constants, $c_1, c_2, \ldots$ Call the resulting vocabulary Σ'. We have to prove that our hypothesis remains valid:

> **Claim:** Δ remains consistent when considered as a set of expressions over Σ'.
>
> ***Proof:*** Suppose that there is a proof $S = (\phi_1, \ldots, \phi_n)$ of some contradiction from Δ using expressions over Σ'. We can assume that there is an infinity of *variables* not appearing in Δ, call them $x_1, x_2, \ldots$ To see why these variables must exist, first recall that our vocabulary of variables is infinite. Even if they all appear in Δ, we can redefine Δ so that it uses only odd-numbered variables from our set, and so an infinity of variables remains unused.
>
> From the proof S of the contradiction we construct a new proof S', in which every occurrence of constant c_i is replaced by variable x_i. This new proof proves a contradiction in the original vocabulary of Δ, which is absurd because Δ was assumed consistent. □

Having introduced the constants c_i, we add to Δ new expressions (enough to make it "conclusive"), as follows. Consider an enumeration of all expressions over Σ': $\phi_1, \phi_2, \ldots$ We shall define a sequence of successive enhancements of Δ by new expressions, Δ_i, for $i = 0, 1, \ldots$. Define $\Delta_0 = \Delta$, and suppose that $\Delta_1, \ldots, \Delta_{i-1}$ have already been defined. Δ_i depends on Δ_{i-1} and ϕ_i. There are four cases:

Case 1: $\Delta_{i-1} \cup \{\phi_i\}$ is consistent, and ϕ_i is not of the form $\underline{\exists x \psi}$. Then $\Delta_i = \Delta_{i-1} \cup \{\phi_i\}$.

Case 2: $\Delta_{i-1} \cup \{\phi_i\}$ is consistent, and $\phi_i = \underline{\exists x \psi}$. Let c be a constant not appearing in any one of the expressions $\phi_1, \ldots, \phi_{i-1}$ (since we have an infinite supply of constants, such a constant exists). Then we let $\Delta_i = \Delta_{i-1} \cup \{\underline{\exists x \psi}, \underline{\psi[x \leftarrow c]}\}$.

Case 3: $\Delta_{i-1} \cup \{\phi_i\}$ is inconsistent, and ϕ_i is not of the form $\underline{\forall x \psi}$. Then $\Delta_i = \Delta_{i-1} \cup \{\underline{\neg \phi_i}\}$.

Case 4: $\Delta_{i-1} \cup \{\phi_i\}$ is inconsistent, and $\phi_i = \underline{\forall x \psi}$. Let c be a constant not appearing in any one of the expressions $\phi_1, \ldots, \phi_{i-1}$ (since we have an infinite supply of constants, such a constant exists). Then we let $\Delta_i = \Delta_{i-1} \cup \{\underline{\neg \forall x \psi}, \underline{\neg \psi[x \leftarrow c]}\}$.

Notice what these additions do to Δ: In all four cases, we add to it all kinds of compatible expressions, slowly solving the problems of "inconclusiveness" alluded to above; naturally, in cases 3 and 4 we refrain from adding to Δ an

expression that would make it inconsistent, but we incorporate its negation. Finally, in cases 2 and 4, as soon as an expression $\exists x\psi$ is added to Δ, an expression $\psi[x \leftarrow c]$, ready to provide a witness, goes along with it.

Claim: For all $i \geq 0$, Δ_i is consistent.

Proof: Induction on i. For the basis, $\Delta_0 = \Delta$ is definitely consistent. Suppose then that Δ_{i-1} is consistent. If Δ_i was obtained by case 1, then it is clearly consistent. If case 3 prevails, then we know that $\Delta_{i-1} \cup \{\phi_i\}$ is inconsistent, and since Δ_{i-1} is consistent, so is $\Delta_i = \Delta_{i-1} \cup \{\neg\phi_i\}$.

Finally, suppose that, case 4 prevails, and Δ_i is inconsistent (the same argument holds for case 2). Arguing by contradiction, $\Delta_{i-1} \cup \{\neg\psi[x \leftarrow c]\} \vdash \forall x\psi$; but we know that also $\Delta_{i-1} \vdash \neg\forall x\psi$, and so $\Delta_{i-1} \cup \{\neg\psi[x \leftarrow c]\}$ is inconsistent. Arguing by contradiction, $\Delta_{i-1} \vdash \psi[x \leftarrow c]$. However, we know that c does not appear in any expression in Δ_{i-1}, and thus we can replace c by a new variable y throughout this latter proof. Thus, $\Delta_{i-1} \vdash \psi[x \leftarrow y]$. By justified generalization, $\Delta_{i-1} \vdash \forall y\psi[x \leftarrow y]$. We can now apply Example 5.8, and take an alphabetic variant of the latter expression, to obtain $\Delta_{i-1} \vdash \forall x\psi$. But we know that $\Delta_{i-1} \cup \{\forall x\psi\}$ is inconsistent, and thus Δ_{i-1} is inconsistent, a contradiction. □

Define thus Δ' to be the union of all Δ_i's. That is, Δ' contains all expressions that appear in some Δ_i, $i \geq 0$. Δ' contains the original Δ, and has some remarkable properties: First, for any expression ϕ over Σ', either ϕ or $\neg\phi$ is in Δ': We say Δ' is *complete*. Second, for any expression $\exists x\phi$ in Δ', there is in Δ' an expression of the form $\phi[x \leftarrow c]$: We say Δ' is *closed*. Finally, it is easy to see the Δ' is consistent: Any proof of a contradiction would involve finitely many expressions, and thus only expressions in Δ_i, for some i. But we know that Δ_i is consistent, and thus no contradictions can be proved from it.

We are now in a position to apply our original idea of "semantics by syntax." Consider the set T of all terms over Σ'. Define an equivalence relation on this set, as follows: $t \equiv t'$ if and only if $t = t' \in \Delta'$. We shall argue that $\equiv$ is an equivalence relation, that is, it is reflexive, symmetric, and transitive. In proof, these three properties, considered as expressions involving terms t, t', and t'', are either axioms or first-order theorems (recall Example 5.4 and see Problem 5.8.8), and are therefore in Δ', since Δ' is consistent and complete. The equivalence class of term $t \in T$ is denoted $[t]$. Let U be the set of equivalence classes of T under $\equiv$. *U is the universe of our model M.*

We next define the values of the functions and the predicates under M. If f is a k-ary function symbol in Σ and $t_1, \ldots, t_k$ are terms, then $f^M([t_1], \ldots, [t_k]) = [f(t_1, \ldots, t_k)]$. If R is a k-ary relation symbol in Σ and $t_1, \ldots, t_k$ are terms, then $R^M([t_1], \ldots, [t_k])$ if and only if $R(t_1, \ldots, t_k) \in \Delta'$. This completes the definition

of M.

It takes some argument to verify that this model "makes sense," that the definitions of f^M and R^M are independent of the particular representatives for the class $[t_i]$. Suppose that $t_1, \ldots, t_k, t'_1, \ldots, t'_k$ are terms such that $t_i \equiv t'_i$ for all i. Then we claim that $f(t_1, \ldots, t_k) \equiv f(t'_1, \ldots, t'_k)$, and thus $[f(t_1, \ldots, t_k)]$ is indeed independent of the choice of the representatives t_i. The reason is that $t_i \equiv t'_i$ for all i means $t_i = t'_i \in \Delta'$ for all i, and hence $\Delta' \vdash f(t_1, \ldots, t_k) = f(t'_1, \ldots, t'_k)$, with the help of axiom **AX2b**. Hence the latter expression is in Δ', and $f(t_1, \ldots, t_k) \equiv f(t'_1, \ldots, t'_k)$. Also, the same way we establish, by induction on the structure of t, that $t^M = [t]$. As for the definition of R^M: If $t_i \equiv t'_i$ for all i, then $M \models R(t_1, \ldots, t_k)$ if and only if $M \models R(t'_1, \ldots, t'_k)$, through axiom **AX2c**. The following result now finishes the proof of the completeness theorem:

Claim: $M \models \Delta'$.

Proof: We shall show, by induction on the structure of ϕ that $M \models \phi$ if and only if $\phi \in \Delta'$. For the basis, suppose that ϕ is atomic, $\phi = R(t_1, \ldots, t_k)$. By the definition of R^M, we know that $R^M(t_1^M, \ldots, t_k^M)$ if and only if $\phi \in \Delta'$.

Suppose then that $\phi = \neg\psi$. $M \models \phi$ if and only if $M \not\models \psi$. By induction, $M \not\models \psi$ if and only if $\psi \notin \Delta'$, which, by the completeness of Δ' happens if and only if $\phi \in \Delta'$. If $\phi = \psi_1 \vee \psi_2$, then $M \models \phi$ if and only if $M \models \psi_i$ for at least one i, which, by induction, happens if and only if at least one ψ_i is in Δ'. By consistency and completeness, this happens if and only if $\phi \in \Delta'$. Similarly for the case $\phi = \psi_1 \wedge \psi_2$.

Suppose then that $\phi = \forall x \psi$. We claim that $\phi \in \Delta'$ if and only if $\psi[x \leftarrow t] \in \Delta'$ for all terms t (this would complete the proof). Suppose that $\phi \in \Delta'$. Then, for each term t, $\Delta' \vdash \psi[x \leftarrow t]$ (by invoking the axiom $\forall x \psi \Rightarrow \psi[x \leftarrow t]$ and *modus ponens*). Since Δ' is complete, $\psi[x \leftarrow t] \in \Delta$. Conversely, suppose that $\phi \notin \Delta'$. Then $\neg\forall\psi$ was introduced to Δ' during its construction, and $\neg\psi[x \leftarrow c]$ was introduced along with it. Hence, $\psi[x \leftarrow c] \notin \Delta'$. Since c is a term of the language, the proof is complete. □

5.6 CONSEQUENCES OF THE COMPLETENESS THEOREM

Since the completeness and soundness theorems identify valid sentences with first-order theorems, the computational problem VALIDITY (given an expression, is it valid?) is the same as THEOREMHOOD. Hence, by Proposition 5.11:

Corollary 1: VALIDITY is recursively enumerable. □

Another immediate consequence of the completeness theorem is the following important property of first-order logic (the same result for Boolean logic,

much easier to prove, was the object of Problem 4.4.9):

Corollary 2 (The Compactness Theorem): If all *finite* subsets of a set of sentences Δ are satisfiable, then Δ is satisfiable.

Proof: Suppose that Δ is not satisfiable, but all of its finite subsets are. Then, by the completeness theorem, there is a proof of a contradiction from Δ: $\Delta \vdash \phi \wedge \neg\phi$. This proof employs finitely many sentences from Δ (recall our definition of a proof in Section 5.4). Therefore, there is a finite subset of Δ (the one involved in the proof of the contradiction) that is unsatisfiable, contrary to our assumption. □

A first application of the compactness theorem shows that the nonstandard model $\mathbf{N}'$ of number theory (the natural numbers plus a disjoint copy of the integers, introduced in Section 5.2) cannot be differentiated from $\mathbf{N}$ by first-order sentences.

Corollary 3: If Δ is a set of first-order sentences such that $\mathbf{N} \models \Delta$, then there is a model $\mathbf{N}'$ such that $\mathbf{N}' \models \Delta$, and the universe of $\mathbf{N}'$ is a proper superset of the universe of $\mathbf{N}$.

Proof: Consider the sentences $\phi_i = \exists x((x \neq 0) \wedge (x \neq 1) \wedge \ldots (x \neq i))$, and the set $\Delta \cup \{\phi_i : i \geq 0\}$. We claim that this set is consistent. Because, if it were not, it would have a finite subset that is inconsistent. This finite subset contains only finitely many of the ϕ_i sentences. But $\mathbf{N}$ obviously satisfies both Δ and any finite subset of these sentences. So, there is a model of $\Delta \cup \{\phi_i : i \geq 0\}$, and the universe of this model must be a strict superset of the universe of $\mathbf{N}$. □

The proof of the completeness theorem establishes the following basic fact about models:

Corollary 4: If a sentence has a model, it has a countable model.

Proof: The model M constructed in the proof of the completeness theorem is countable, since the vocabulary Σ' is countable. □

But, of course, a countable model can be either finite or infinite. The model M in the proof of the completeness theorem is in general infinite. Is it true that all sentences have a countably infinite model? The answer is "no." Some sentences, such as $\forall x \forall y(x = y)$ and $\exists x \exists y \forall z(z = x \vee z = y)$ have no infinite models. However, this is achieved by specifying an upper bound on the cardinality of any model (one for the former, two for the latter). Our next result states in essence that *this is the only way* for a sentence to avoid satisfaction by infinite models.

Corollary 5 (Löwenheim-Skolem Theorem): If sentence ϕ has finite models of arbitrary large cardinality, then it has an infinite model.

Proof: Consider the sentence $\psi_k = \exists x_1 \ldots \exists x_k \bigwedge_{1 \leq i < j \leq k} \neg(x_i = x_j)$, where $k > 1$ is an integer. Obviously, ψ_k states that there are at least k distinct

elements in the universe; it cannot be satisfied by a model with a universe containing $k-1$ or fewer elements, and any model with k or more elements satisfies it.

Let us assume, for the sake of contradiction, that ϕ has arbitrarily large models, but no infinite model. Consider thus the set of sentences $\Delta = \{\phi\} \cup \{\psi_k : k = 2, 3, \ldots\}$. If Δ has a model M, then M cannot be finite (because, if it were, and had k elements, it would not satisfy ψ_{k+1}), and cannot be infinite (it would satisfy ϕ). So, Δ has no model.

By the compactness theorem, there is a finite set $D \subset \Delta$ that has no model. This subset must contain ϕ (otherwise, any sufficiently large model would satisfy all ψ_k's in D). Suppose then that k is the largest integer such that $\psi_k \in D$. By our hypothesis, ϕ has a finite model of cardinality larger than k. It follows that this model satisfies all sentences in D, a contradiction. □

Finally, let us use the Löwenheim-Skolem theorem to prove that a limitation in the expressibility of first-order logic. In Section 5.3 we gave several examples of interesting properties of graphs (outdegree one, symmetry, transitivity, etc.) that can be expressed in first-order logic. We also showed that any graph property that can be expressed by a sentence ϕ is easy to check (the corresponding computational problem, called ϕ-GRAPHS, has a polynomial algorithm). A natural question arises: Are all polynomially checkable graph properties thus expressible?

The answer is "no," and, the problem that establishes it is REACHABILITY (given a graph G and two nodes x and y of G, is there a path from x to y?).

Corollary 6: There is no first-order expression ϕ (with two free variables x and y) such that ϕ-GRAPHS is the same as REACHABILITY.

Proof: Suppose there were such an expression ϕ, and consider the sentence $\psi_0 = \forall x \forall y \phi$. It states that the graph G is *strongly connected*, that is, all nodes are reachable from all nodes. Consider then the conjunction of ψ_0 with the sentences $\psi_1 = (\forall x \exists y G(x,y) \wedge \forall x \forall y \forall z((G(x,y) \wedge G(x,z)) \Rightarrow y = z))$ and $\psi_1' = (\forall x \exists y G(y,x) \wedge \forall x \forall y \forall z((G(y,x) \wedge G(z,x)) \Rightarrow y = z))$. Sentence ψ_1 states that every node of G has outdegree one, and ψ_1' states that every node of G has indegree one. Thus, the conjunction $\psi = \psi_0 \wedge \psi_1 \wedge \psi_1'$ states that the graph is strongly connected, and all nodes have both indegree and outdegree one. Such graphs have a name: They are *cycles* (see Figure 5.5).

Obviously, there are finite cycles with as many nodes as desired; this means that ψ has arbitrary large finite models. Therefore, by the Löwenheim-Skolem theorem, it has an infinite model, call it G_∞. A contradiction is near: Infinite cycles do not exist! Specifically, let us consider a node of G_∞, call it node 0. Node 0 has a single edge out of it, going to node 1, node 1 has a single edge out of it, going to node 2, from there to node 3, and so on. If we continue this way, we eventually encounter all nodes reachable by a path from node 0. Since G_∞ is

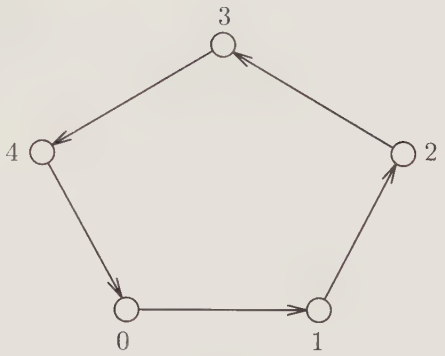


Figure 5-5. A cycle.

strongly connected, these nodes must include all nodes of the graph. However, 0 has indegree one, and so there must be a node, call it j, such that there is an edge from j to 0. This is a contradiction: The "infinite" cycle is indeed finite. □

The inexpressibility of reachability by first-order logic is very interesting, for several reasons: First, it is a nontrivial *impossibility* theorem, and impossibility theorems are precisely the kinds of results we would have liked to prove in complexity theory (and are still so far from proving...). Besides, this impossibility result has motivated the study of appropriate extensions of first-order logic, and ultimately resulted in another important facet of the close identification between complexity and logic. These added capabilities are the last subject of this chapter.

5.7 SECOND-ORDER LOGIC

What "features" can we add to first-order logic so that REACHABILITY can be expressed? Intuitively, what we want to state in REACHABILITY is that "there is a path from x to y." On the surface, a statement that starts with "there is" seems ideal for expressibility in first-order logic, with its $\exists$ quantifier. But of course there is a catch: Since our quantifiers are immediately followed by variables like x, and variables stand for individual nodes, first-order logic only allows us to state "there is a node x...". Evidently, what is needed here is the ability to start our sentence with an existential quantifier over more complex objects.

Definition 5.4: An expression of *existential second-order logic* over a vocabulary $\Sigma = (\Phi, \Pi, r)$ is of the form $\underline{\exists P \phi}$, where ϕ is a first-order expression over the vocabulary $\Sigma' = (\Phi, \Pi \cup \{P\}, r)$. That is, $P \notin \Pi$ is a new relational symbol of arity $r(P)$. Naturally, P can be mentioned in ϕ; intuitively, expression $\underline{\exists P \phi}$

says that there is a relation P such that ϕ holds.

The semantics of an expression of second-order logic captures this intuition: A model M appropriate for Σ satisfies $\exists P\phi$ if there is a relation $P^M \subseteq (U^M)^{r(P)}$ such that M, augmented with P^M to comprise a model appropriate for Σ', satisfies ϕ. □

Example 5.9: Consider the second-order expression (in the vocabulary of number theory) $\phi = \exists P \forall x((P(x) \vee P(x+1)) \wedge \neg(P(x) \wedge P(x+1)))$. It asserts the existence of a set P such that for all x either $x \in P$ or $x+1 \in P$ but not both. ϕ is satisfied by $\mathbf{N}$, the standard model of number theory: Just take $P^{\mathbf{N}}$ to be the set of even numbers. □

Example 5.10: The sentence $\exists P \forall x \forall y(G(x,y) \Rightarrow P(x,y))$ in the vocabulary of graph theory asserts the existence of a *subgraph* of graph G. It is a valid sentence, because any graph has at least one subgraph: Itself (not to mention the empty subgraph...). □

Example 5.11: Our next expression of second-order logic captures graph reachability. More precisely, it expresses *unreachability*, the complement of reachability:

$$\phi(x,y) = \exists P(\forall u \forall v \forall w((P(u,u)) \wedge (G(u,v) \Rightarrow P(u,v)) \wedge$$

$$\wedge((P(u,v) \wedge P(v,w)) \Rightarrow P(u,w)) \wedge \neg P(x,y)))$$

$\phi(x,y)$ states that there is a graph P which contains G as a subgraph, is reflexive and transitive, and furthermore in this graph there is no edge from x to y. But it is easy to see any P that satisfies the first three conditions must contain an edge between any two nodes of G that are reachable (in other words, it must contain the *reflexive-transitive closure* of G). So, the fact that $\neg P(x,y)$ implies that there is no path from x to y in G: $\phi(x,y)$-GRAPHS is precisely the complement of REACHABILITY.

We can express REACHABILITY itself with a little more work, using expressions like those employed in the next example for a different purpose, see Problem 5.8.13 (we cannot do this by simply negating ϕ: Existential second-order logic is *not* obviously closed under negation). Therefore, existential second-order logic succeeds in the quest that motivated this section. □

Example 5.12: In fact, existential second-order logic is much more powerful than that. It can be used to express graph-theoretic properties which, unlike REACHABILITY, have no known polynomial-time algorithm. For example, consider the problem HAMILTON PATH: Given a graph, is there a path that visits each node exactly once? For example, the graph in Figure 5.6 has a Hamilton path, shown with heavy lines. This is a proverbially hard problem (whose kinship to the TSP, recall Section 1.3, is almost too obvious). Currently no

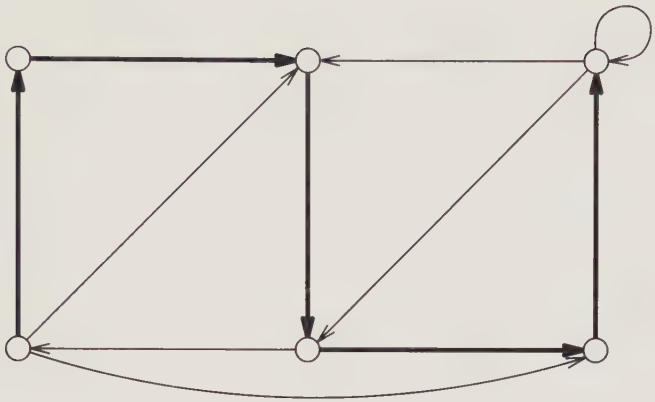


Figure 5-6. Hamilton path.

polynomial-time algorithm is known for telling whether a graph has a Hamilton path.

Interestingly, the following sentence describes graphs with a Hamilton path: $\psi = \exists P \chi$. Here χ will require that P be a *linear order* on the nodes of G, that is, a binary relationship isomorphic to $<$ on the nodes of G (without loss of generality, these nodes are $\{1, 2, \ldots, n\}$), such that consecutive nodes are connected in G. χ must require several things: First, all distinct nodes of G are comparable by P:

$$\forall x \forall y((P(x, y) \vee P(y, x) \vee x = y). \tag{1}$$

Next, P must be transitive but not reflexive:

$$\forall x \forall y \forall z((\neg P(x, x)) \wedge ((P(x, y) \wedge P(y, z)) \Rightarrow P(x, z)).$$

Finally, any two *consecutive* nodes in P must be adjacent in G:

$$\forall x \forall y((P(x, y) \wedge \forall z(\neg P(x, z) \vee \neg P(z, y))) \Rightarrow G(x, y)).$$

It is easy to check that ψ-GRAPHS is the same as HAMILTON PATH. This is because any P that has these properties must be a linear order, any two consecutive elements of which are adjacent in G—that is, it must be a Hamilton path. □

The last example suggests that, in our effort to extend first-order logic so that it captures simple properties like graph reachability, we got a little carried away: Existential second-order logic is much too powerful, as it can also express very complex properties, like the existence of Hamiltonian paths, not believed to be in **P**. However, it is easy to show the following:

Theorem 5.8: For any existential second-order expression $\exists P\phi$, the problem $\exists P\phi$-GRAPHS is in **NP**.

Proof: Given any graph $G = (V, E)$ with n nodes, a nondeterministic Turing machine can "guess" a relation $P^M \subseteq V^{r(P)}$ such that G augmented with P^M satisfies ϕ, if such relation exists. The machine can then go on to test that indeed M satisfies the first-order expression ϕ deterministically in polynomial time (using the polynomial algorithm in Theorem 5.1). The overall elapsed time for guessing and checking is polynomial, because there are at most $n^{r(P)}$ elements of P^M to guess. □

Let us now look back at the two expressions, one for (UN)REACHABILITY the other for HAMILTON PATH, to see why the former corresponds to an easy problem, and the latter to a hard one. The expression $\phi(x, y)$ for UNREACHABILITY is in prenex normal form with only universal first-order quantifiers, and with matrix in conjunctive normal form. What is more important, suppose that we delete from the clauses of the matrix anything that is not an atomic expression involving P. The three resulting clauses are these:

$$(P(u, u)), \quad \neg(P(x, y)), \quad (\neg P(x, y) \vee \neg P(y, z) \vee P(x, z)).$$

Notice that all three of these clauses have *at most one unnegated atomic formula involving* P. This recalls the *Horn clauses* in Boolean logic (Theorem 4.2). Let us call an expression in existential second-order logic a *Horn expression* if it is in prenex form with only universal first-order quantifiers, and its matrix is the conjunction of clauses, each of which contains at most one unnegated atomic formula that involves P, the second-order relation symbol. Obviously, ϕ above is a Horn expression. In contrast, expression ψ for HAMILTON PATH contains a host of violations of the Horn form. If it is brought into prenex form there will be existential quantifiers. And its clause (1) is inherently non-Horn.

The following result then explains the difference between ϕ and ψ:

Theorem 5.9: For any Horn existential second-order expression $\exists P\phi$, the problem $\exists P\phi$-GRAPHS is in **P**.

Proof: Let

$$\exists P\phi = \exists P \forall x_1 \ldots \forall x_k \eta, \tag{2}$$

where η is the conjunction of Horn clauses and the arity of P is r. Suppose that we are given a graph G on n vertices (without loss of generality, the vertices of G are $\{1, 2, \ldots, n\}$), and we are asked whether G is a "yes" instance of $\exists P\phi$-GRAPHS. That is, we are asked whether there is a relation $P \subseteq \{1, 2, \ldots, n\}^r$ such that ϕ is satisfied.

Since η must hold for all combinations of values for the x_i's, and the x_i's take values in $\{1, 2, \ldots, n\}$, we can rewrite (2) as a huge conjunction of different

variants of η, where in each we have substituted all possible values for the x_i's:

$$\bigwedge_{v_1,\ldots,v_k=1}^{n} \eta[x_1 \leftarrow v_1, \ldots, x_k \leftarrow v_k]. \tag{3}$$

This expression contains exactly hn^k clauses, where h is the number of clauses in η.

Now, the atomic expressions in each clause of (3) can be of only three kinds: Either $G(v_i, v_j)$, or $v_i = v_j$, or $P(v_{i_1}, \ldots, v_{i_r})$. However, the first two kinds can be easily evaluated to **true** or **false** and disposed of (remember, the v_i's are actual vertices of G, and we have G). If a literal is found to be **false**, it is deleted from its clause; if it is found to be **true**, its clause is deleted. And if we end up with an empty clause, we conclude that the expression is not satisfiable and G does not satisfy ϕ. Therefore, unless we are done, we are left with a conjunction of at most hn^k clauses, each of which is the disjunction of atomic expressions of the form $P(v_{i_1}, \ldots, v_{i_r})$ and their negations.

We need one last idea: Each of these atomic expressions can be independently **true** or **false**. So, *why not replace each by a different Boolean variable.* That is, we consistently replace each occurrence of $P(v_{i_1}, \ldots, v_{i_r})$ by the new Boolean variable $x^{v_{i_1},\ldots,v_{i_r}}$. The result is a Boolean expression F. It follows immediately from our construction that F is satisfiable if and only if there is a P such that P with G satisfy ϕ.

Finally, since η was the conjunction of Horn second-order clauses, F *is a Horn Boolean expression* with at most hn^k clauses and at most n^r variables. Therefore, we can solve the satisfiability problem of F —and thus the given instance of $\exists P\phi$-GRAPHS – by our polynomial-time algorithm for HORNSAT (Theorem 4.2). □

5.8 NOTES, REFERENCES, AND PROBLEMS

5.8.1 Logic is an extensive and deep branch of mathematics, and these three chapters (4, 5, and 6) only touch on some of its rudiments—in particular, those that seem most pertinent to our project of understanding complexity. For more ambitious introductions see, for example,

- H. B. Enderton *A Mathematical Introduction to Logic*, Academic Press, New York, 1972, and
- J. R. Schoenfield *Mathematical Logic*, Addison-Wesley, Reading, Massachusetts, 1967, and perhaps the last chapter in
- H. R. Lewis, C. H. Papadimitriou *Elements of the Theory of Computation*, Prentice-Hall, Englewood Cliffs, 1981.

5.8.2 Problem: Show that there is a polynomial algorithm for testing a graph for each of the three first-order properties in Section 5.2: Outdegree one, symmetry, and transitivity. How is the exponent of your algorithm related to the logical expression that defines the corresponding problem?

5.8.3 Problem: Show by induction on ϕ that ϕ-GRAPHS can be tested in logarithmic space (Corollary to Theorem 5.1). How does the number of quantifiers affect the complexity of your algorithm?

5.8.4 Problem: Prove the four equivalences in Proposition 5.10 by arguing in terms of models satisfying the two sides.

5.8.5 Problem: Prove Theorem 5.2 stating that any expression can be transformed into an equivalent one in prenex normal form.

5.8.6 Problem: Show that there is an algorithm which, given an expression, determines whether it is one of the axioms in Figure 5.4. What is the complexity of your algorithm?

5.8.7 Problem: We have a vocabulary with just one relation LOVES (and $=$, of course), and only two constants, ME and MYBABY. Write an expression ϕ that corresponds to the following oldie:

Everybody loves my baby,
my baby loves nobody but me...

(a) Argue that the following expression is valid: $\phi \Rightarrow$MYBABY=ME.
(b) Show that $\{\phi\} \vdash$MYBABY=ME. Give the proof using our axiom system. (This unintended conclusion shows the dangers of naively translating natural language utterances into first-order logic.)
(c) Write a more careful version of ϕ, one not susceptible to this criticism.

5.8.8 Problem: For each of these expressions, if it is valid supply a line-by-line proof from our axiom system. perhaps using deduction, contradiction, and justified generalization; if it is invalid, provide a model that falsifies it:

(a) $\forall x \forall y \forall z((x = y \wedge y = z) \Rightarrow x = z)$.
(b) $\exists y \forall x(x = y + 1) \Rightarrow \forall w \forall z(w = z)$.
(c) $\forall y \exists x(x = y + 1) \Rightarrow \forall w \forall z(w = z)$.
(d) $\forall x \exists y G(x, y) \Rightarrow \exists y \forall x G(x, y)$.
(e) $\forall x \exists y G(x, y) \Leftrightarrow \exists y \forall x G(x, y)$.
(f) $\forall x \phi \Leftrightarrow \forall y \phi[y \leftarrow x]$, where y does not appear in ϕ.

5.8.9 The completeness theorem was proved by Kurt Gödel

- K. Gödel "Die Vollständigkeit der Axiome der Logischen Funktionenkalküls" (The completeness of the axioms of the logical function calculus), *Monat. Math. Physik, 37*, pp. 349 360, 1930.

although our proof follows that by Leon Henkin

- L. Henkin "The completeness of first-order functional calculus," *J. Symb. Logic, 14*, pp., 159–166, 1949.

5.8.10 Problem: Show that the second form of the completeness theorem implies the first form.

5.8.11 Corollary 3, the necessity of non-standard models of number theory, is due to Thoralf Skolem, see

- T. Skolem "Über die Unmöglichkeit einer vollständingen Charakterisierung der Zahlenreihe mittels eines endlichen Axiomsystems" (On the impossibility of the complete characterization of number theory in terms of a finite axiom system), *Norsk Mathematisk Forenings Skrifter, 2,10* , pp. 73–82, 1933.

5.8.12 Herbrand's theorem. There is another important result, besides the completeness theorem, that provides a syntactic characterization of validity in first-order logic (and a proof of the results in Section 5.7), due to Jacques Herbrand; see

- J. Herbrand "Sur la théorie de la démonstration" (On the theory of proof), *Comptes Rend. Acad. des Sciences, Paris, 186*, pp. 1274–1276, 1928.

The result assumes that our language has no equality (although equality can be added with a little more work). Consider a sentence ϕ in prenex normal form. Replace all occurrences of each existentially quantified variable x by a new function symbol f_x, *with arity equal to the number of universal quantifiers before* $\exists x$. The arguments of f_x in all its occurrences are precisely the universally quantified variables that precede $\exists x$. Call the matrix of the resulting sentence ϕ^*.

For example, if ϕ is the expression $\exists x \forall y \exists z(x + 1 < y + z \wedge y < z)$, then ϕ^* is $f_x + 1 < y + f_z(y) \wedge y < f_z(y)$. Notice that, for example, $f_z(y)$ captures the intuitive notion of "there is a z depending on y;" f_x is a constant. Consider the set T of all variable-free terms over our new vocabulary (including the f_x's; if our vocabulary has no constant symbol, then T would be empty, so in this case we add the constant 1 to the vocabulary). Suppose that the variables of ϕ^* (the universally quantified variables of ϕ) are $x_1, \ldots, x_n$. The *Herbrand expansion* of ϕ is the infinite set of all variable-free expressions of the form $\phi[x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n]$, *for all* $t_1, \ldots, t_n \in T$. For example, the Herbrand expansion of the expression above starts like this:

$$\{f_x + 1 < 1 + f_z(1) \wedge 1 < f_z(1), f_x + 1 < f_x + f_z(f_x) \wedge f_x < f_z(f_x),$$
$$f_x + 1 < f_z(1) + f_z(f_z(1)) \wedge f_z(1) < f_z(f_z(1)), \ldots\}.$$

(a) Prove *Herbrand's theorem:* ϕ has a model if and only if the Herbrand expansion of ϕ is satisfiable.

That is, first-order satisfiability is reduced to satisfiability of a countably infinite set of "essentially Boolean" expressions.

(b) Show Corollary 2 of the completeness theorem (the compactness theorem) from Herbrand's theorem and the compactness theorem for Boolean expressions (Problem 4.4.9).

5.8.13 Problem: (a) Give an existential second-order expression for REACHABILITY. (Use the construction in Example 5.12, only do not require that all nodes be in the linear order. Then require that the first node be x, and the last y.)

(b) Give an existential second-order expression for UNREACHABILITY *in which the second-order relation symbol is unary.*

It turns out that the binary existential second-order relation symbol for REACHABILITY used in (a) is *necessary,* see

- R. Fagin "Monadic generalized spectra," *Zeitschrift für Math. Logik und Grund. der Math.,21,* pp. 123–134, 1975; see also
- R. Fagin, L. Stockmeyer, and M. Vardi "A simple proof that connectivity separates existential and universal monadic second-order logics over finite structures," Research Report RJ 8647 (77734), IBM, 1992.

5.8.14 Fixpoint logic provides an alternative to Horn second-order logic for capturing a modest set of graph-theoretic properties. Consider a first-order expression ϕ, and a relation symbol P in ϕ. We say that P *appears positively in* ϕ if there is no occurrence of P within any $\neg\phi'$ subexpression of ϕ (we assume that there are no $\Rightarrow$ connectives in ϕ, and thus no "hidden" $\neg$ symbols).

(a) Suppose that P occurs positively in ϕ, that M and M' are models appropriate for ϕ that are identical in all respects except that $P^M \subseteq P^{M'}$, and that $M \models \phi$. Then $M' \models \phi$.

A *fixpoint expression* over a vocabulary Σ is an expression of the form "$\phi : P = \psi^*$", where (a) ϕ is a first-order expression over Σ, (b) ψ is a first-order expression over Σ with r free variables, say $x_1, \ldots, x_r$, and (c) P an r-ary relation symbol in the vocabulary, appearing positively in ψ. In other words, fixpoint logic simply annotates a first-order expression ϕ with the mysterious statement "$P = \psi^*$."

To define the semantics of fixpoint logic, let "$\phi : P = \psi^*$" be a fixpoint expression. Models appropriate for ϕ (and therefore for ψ, recall that ϕ and ψ are over the same vocabulary) will be represented as (M, F), where $F = P^M$ is the value taken by the special r-ary symbol P. We say that (M, F) is a *P-minimal model* of some first-order expression χ if $(M, F) \models \chi$ *and there is no proper subset F' of F such that* $(M, F') \models \chi$. Finally, (M, F) is said to satisfy "$\phi : P = \psi^*$" if it satisfies ϕ and furthermore it is a P-minimal model of the expression $P(x_1, \ldots, x_r) \Leftrightarrow \psi$.

(b) Show that "$\exists z \neg P(z) : P = \underline{(x = 0 \vee \exists y(P(y) \wedge x = y + 2))^*}$" is satisfied by the standard model $\mathbf{N}$ of arithmetic, if P is taken to be the set $\{0, 2, \ldots\}$ of all even integers.

(c) Consider the fixpoint expression

$$P(x, y) : P = \underline{((x = y) \vee \exists z(P(x, z) \wedge G(z, y)))^*}.$$

Show that model (G, F) satisfies this expression if and only if F is the reflexive-transitive closure of G; that is, $F(x, y)$ if and only if there is a path from x to y.

If "$\phi : P = \psi^*$" is a fixpoint expression on the vocabulary of graph theory, $(\phi : P = \psi^*)$-GRAPHS is the problem of telling, given a graph G, whether there is a relation F of the appropriate arity such that (G, F) satisfy ϕ, and (G, F) is a P-minimal model of $P \Leftrightarrow \psi$). For fixpoint expression in (c) above, $(\phi : P = \psi^*)$-GRAPHS is precisely REACHABILITY.

Consider the following algorithm for $(\phi : P = \psi^*)$-GRAPHS: We build F in stages, starting from $F_0 = \emptyset$; at each stage we add to F only the r-tuples that must be in any model of $P \Leftrightarrow \psi$ (compare with the proof of Theorem 4.2). That is, at stage $i > 0$, F_i is the set of all these r-tuples of nodes of G $(v_1, \ldots, v_r)$ such that $(G, F_{i-1}) \models \psi[x_1 \leftarrow v_1, \ldots, x_r \leftarrow v_r]$.

(d) Show that $F_{i-1} \subseteq F_i$.

(e) Suppose that (G, F) is a model of $P(x_1, \ldots, x_r) \Leftrightarrow \psi$. Then show that $F_i \subseteq F$, for every $i \geq 0$.

(f) Show that after polynomially many stages we obtain the *unique minimal model* F *of* ψ. We can then test in polynomial time whether G with F satisfies ϕ.

(g) Conclude that $(\phi : P = \psi^*)$-GRAPHS is in $\mathbf{P}$.

(h) What are the F_i's when this algorithm is applied to the fixpoint expression for REACHABILITY above?

5.8.15 Problem: (a) Suppose that your vocabulary has only one binary relational symbol, CANFOOL(p, t) (intuitive meaning: "You can fool person p at time t"), and no function and constant symbols. Write an expression in first-order logic which captures the famous quotation by Abraham Lincoln:

You can fool some people all of the time, and all of the people some of the time, but you cannot fool all of the people all of the time.

(b) Now look back at the expression you wrote. It is probably either the *pessimistic* version, in which at least one person can be fooled all of the time, and there is a circumstance in which all people can be fooled, or the *optimistic* version, which leaves open the possibility that no person can be fooled all the time, but different people may be fooled at each time (and similarly for the "all of the people" part). Write the second expression.

(c) One of the two expressions logically implies the other. Give a formal proof from the axioms that it does.

CHAPTER 6

UNDECIDABILITY IN LOGIC

We can express in logic statements about computation. As we already know, such power is ominous; it manifests itself most clearly in the case of number theory.

6.1 AXIOMS FOR NUMBER THEORY

If you had to write down the basic properties of the whole numbers, what would they be? In Figure 6.1 we show some well-known sentences satisfied by model **N**, the nonnegative integers. We use the abbreviation $x \leq y$ to stand for $x = y \vee x < y$, and $x \neq y$ to stand for $\neg(x = y)$. In **NT14**, we have used the relation symbol $\mathrm{mod}(x, y, z)$ as an abbreviation for the expression $\exists w(x = (y \times w) + z \wedge z < y)$ (notice that, indeed, the expression states that z is the remainder of the division of x by y). On the other hand, the expression $\exists z(x = (y \times w) + z \wedge z < y)$ (notice the difference) can be called $\mathrm{div}(x, y, w)$.

Obviously, these sentences are true properties of the integers. Axioms **NT1**, **NT2**, and **NT3** establish the function σ as a graph with indegree and outdegree one, except for 0, where it has indegree zero. Axioms **NT4** and **NT5** are essentially an inductive definition of addition. Similarly, axioms **NT6** and **NT7** define multiplication, and axioms **NT8** and **NT9** define exponentiation. Axioms **NT10**, **NT11**, **NT12**, and **NT13** are useful properties of inequality. Finally, axiom **NT14** states that mod is a function. We shall denote by **NT** the conjunction **NT1** $\wedge$ **NT2** $\wedge \ldots \wedge$ **NT14**.

What is the power of this set of axioms? We know it is sound (that is, it contains no false property of **N**); ideally, we would like it also to be *complete*, that is, able to prove all true properties of **N**. Our ultimate result in this

NT1:	$\forall x(\sigma(x) \neq 0)$
NT2:	$\forall x \forall y(\sigma(x) = \sigma(y) \Rightarrow x = y)$
NT3:	$\forall x(x = 0 \vee \exists y \sigma(y) = x)$
NT4:	$\forall x(x + 0 = x)$
NT5:	$\forall x \forall y(x + \sigma(y) = \sigma(x + y))$
NT6:	$\forall x(x \times 0 = 0)$
NT7:	$\forall \hat{x} \forall y(x \times \sigma(y) = (x \times y) + x)$
NT8:	$\forall x(x \uparrow 0 = \sigma(0))$
NT9:	$\forall x \forall y(x \uparrow \sigma(y) = (x \uparrow y) \times x)$
NT10:	$\forall x(x < \sigma(x))$
NT11:	$\forall x \forall y(x < y \Rightarrow (\sigma(x) \leq y)$
NT12:	$\forall x \forall y(\neg(x < y) \Leftrightarrow y \leq x)$
NT13:	$\forall x \forall y \forall z(((x < y) \wedge (y < z)) \Rightarrow x < z)$
NT14:	$\forall x \forall y \forall z \forall z'(\mathrm{mod}(x, y, z) \wedge \mathrm{mod}(x, y, z') \Rightarrow z = z')$

Figure 6.1. Nonlogical axioms for number theory.

chapter is that such sound and complete axiom system for **N** *does not exist;* so there are many true properties of the integers that cannot be proved from **NT** (a simple example is the commutativity of addition, see Example 6.3 below). However, we shall prove in this section that **NT** is complete *for a quite general set of sentences.* That is, we shall define a subclass of the sentences in the vocabulary of Number Theory (a "syntactic fragment" of number theory) such that for any sentence ϕ in the class either $\mathbf{NT} \vdash \phi$ (in the case that ϕ is a true property of the whole numbers), or $\mathbf{NT} \vdash \neg\phi$ (in the case $\mathbf{N} \not\models \phi$). We have to start from very humble beginnings:

Example 6.1: The reader is not likely to be overly impressed by this, but $\{\mathbf{NT}\} \vdash 1 < 1 + 1$. Here is how this proof goes:

We first show that $\{\mathbf{NT}\} \vdash \forall x(\sigma(x) = x + 1)$. This follows from **NT5**, by specializing to $y = 0$, and using **NT4** to replace $x + 0$ by x. Having established $\forall x(x + 1 = \sigma(x))$, $\forall x(\sigma(x) = x + 1)$ follows.

We then use **NT10**, specialized at $x = \sigma(0) = 1$, to get $1 < \sigma(1)$, which, together with the "lemma" $\forall x(\sigma(x) = x + 1)$ proved above, yields the desired result $1 < 1 + 1$.

Incidentally, notice the liberating influence the completeness theorem has had on our proof style: Since we now know that all valid implications are provable, to establish that $\{\mathbf{NT}\} \vdash \phi$ we must simply argue that ϕ is a true property of the integers *if we pretend that the only facts we know about the integers are axioms* **NT1** *through* **NT14**. All valid manipulations of first-order

expressions are allowed. □

In fact, we can show that the above example is part of a far more general pattern. A *variable-free* sentence is one that has no variable occurrences (free *or* bound). It turns out that **NT** is enough to prove all true variable-free sentences, and disprove all false ones:

Theorem 6.1: Suppose that ϕ is a variable-free sentence. Then $\mathbf{N} \models \phi$ if and only if $\{\mathbf{NT}\} \vdash \phi$.

Proof: Any variable-free sentence is the Boolean combination of expressions of one of these forms: $t = t'$ and $t < t'$. It therefore suffices to prove the theorem for these two kinds of expressions.

Suppose first that both t and t' are "numbers," that is, terms of the form $\sigma(\sigma(\ldots\sigma(0)\ldots))$. Then $t = t'$ is trivial to prove if correct. To prove a correct inequality $t < t'$ in the case of "numbers" we use **NT10** to prove a sequence of inequalities $t < \sigma(t)$, $\sigma(t) < \sigma(\sigma(t))$, etc., up to t', and then use **NT13**. So, we know how to prove from **NT** true equalities and inequalities involving only "numbers."

So, suppose that t and t' are general variable-free terms, such as $t = (3 + (2 \times 2)) \uparrow (2 \times 1)$. Obviously, any such term t has a *value*. That is, there is a "number" t_0 such that $\mathbf{N} \models t = t_0$ (in the above example $t_0 = 49$). If we can show $\{\mathbf{NT}\} \vdash t = t_0$ and $\{\mathbf{NT}\} \vdash t' = t'_0$, we can replace these "numbers" in the equality or inequality to be proved, and use the method in the previous paragraph. So, it remains to show that for any variable-free term t $\{\mathbf{NT}\} \vdash t = t_0$, where t_0 is the "value" of t; that is, the conversion of any variable-free term to its value can be proved correct within **NT**.

We show this by induction on the structure of t. If $t = 0$, then it is immediate. If $t = \sigma(t')$, then by induction we first show $t = t'_0$, from which $\sigma(t') = \sigma(t'_0)$ follows (and $t_0 = \sigma(t'_0)$). Finally, if $t = t_1 \circ t_2$ for some $\circ \in \{+, \times, \uparrow\}$, then we first prove, by induction, $\{\mathbf{NT}\} \vdash t_1 = t_{10}$ and $\{\mathbf{NT}\} \vdash t_2 = t_{20}$ for appropriate "numbers." Then we repeatedly apply to $t = t_{10} \circ t_{20}$ the axioms that define the operations (**NT9** for $\uparrow$, **NT7** for $\times$, **NT5** for $+$) until the base axioms (**NT8**, **NT6**, and **NT4**, respectively) become applicable. It is easy to see that ultimately the expression will be reduced to its value. □

Notice that also, for any variable-free sentence ϕ, if $\mathbf{N} \not\models \phi$ then $\{\mathbf{NT}\} \vdash \neg\phi$. This is immediate from Theorem 6.1 because, if $\mathbf{N} \not\models \phi$ then $\mathbf{N} \models \neg\phi$. So, we can prove from **NT** any true property of the integers without quantifiers, and disprove any false such property.

But we can also prove from **NT** expressions involving quantifiers. We have already done so in Example 6.1, where we proved $\forall x(\sigma(x) = x + 1)$, by having this sentence "inherit" one of the two universal quantifiers of **NT5**. For existential quantifiers there is a quite general proof technique.

Example 6.2: Let us prove from **NT** that the Diophantine equation $x^x + x^2 -$

$4x = 0$ has an integer solution (no, it is not 0). This statement is expressed as $\exists x((x \uparrow x) + (x \uparrow 2) = 4 \times x)$. Let us call the matrix of this expression ϕ. To prove that $\{\mathbf{NT}\} \vdash \exists x\phi$, we simply have to establish that $\phi[x \leftarrow \underline{2}]$ follows from **NT**. And we know how to do that, by Theorem 6.1, since $\mathbf{N} \models \phi[x \leftarrow \underline{2}]$. □

Example 6.3: Despite Theorem 6.1 and the example above, **NT1** through **NT14** constitute a very weak attempt at axiomatizing number theory. For example, $\{\mathbf{NT}\} \not\vdash \forall x \forall y(x + y = y + x)$. A proof of this sentence (the commutativity property of addition) requires *induction*, that is, the use of the following group of axioms (one for every expression $\phi(x)$ with a free variable x, in the style of our axioms for first-order logic, recall Figure 5.4):

$$\mathbf{NT15}: (\phi(0) \wedge \forall x(\phi(x) \Rightarrow \phi(\sigma(x)))) \Rightarrow \forall y \phi(y).$$

We have refrained from adding **NT15** to our axiom system simply because we wanted to keep it finite, so we can talk about a single sentence **NT**. □

Evidently, **NT** is especially weak at proving universally quantified sentences. However, there is a special form of universal quantifier that can be handled. We use the notation $(\forall x < t)\phi$, where t is a term, as an abbreviation for $\forall x((x < t) \Rightarrow \phi)$; similarly for $(\exists x < t)\phi$. We call these *bounded quantifiers*. When all quantifiers (bounded and unbounded) precede the rest of the expression, we say that the expression is in *bounded prenex form*. So, sentence $(\forall x < 9)\exists y(\forall z < 2 \times y)(x + z + 10 < 4 \times y)$ is in bounded prenex form (although it is not in prenex form if we expand the abbreviation $\forall x < 9$). We call a sentence *bounded* if all of its universal quantifiers are bounded, and the sentence is in bounded prenex form. For example, the sentence above is bounded. We can now spell out the power of **NT**:

Theorem 6.2: Suppose that ϕ is a bounded sentence. Then $\mathbf{N} \models \phi$ if and only if $\{\mathbf{NT}\} \vdash \phi$.

Proof: We shall prove by induction on the number k of quantifiers of ϕ that $\mathbf{N} \models \phi$ implies $\{\mathbf{NT}\} \vdash \phi$ (the other direction is obvious, since **NT** is sound). First, if k is zero, ϕ is a variable-free sentence, and hence the result follows from Theorem 6.1. Suppose that $\phi = \exists x\psi$. Since $\mathbf{N} \models \phi$, there is an integer n such that $\mathbf{N} \models \psi[x \leftarrow \underline{n}]$. By induction, $\{\mathbf{NT}\} \vdash \psi[x \leftarrow \underline{n}]$, and hence $\{\mathbf{NT}\} \vdash \phi$ (notice that this was the trick used in Example 6.2).

Finally, suppose that $\phi = (\forall x < t)\psi$. Then t is a variable-free term (since there are no other bound variables at this position in the expression, and there are no free variables anyway). Hence, by Theorem 6.1 we can assume that t is a number, call it n (otherwise, we prove from **NT** that $t = \underline{n}$, and then replace t by $\underline{n}$). For any integer n, it is easy to prove, by repeated applications of **NT10** and **NT11**, $\chi_1 = \forall x(x < n \Rightarrow (x = 0 \vee x = 1 \vee x = 2 \vee \ldots \vee x = n - 1))$. Substitute now each value j, $0 \leq j < n$, into ψ. By induction, $\{\mathbf{NT}\} \vdash \psi[x \leftarrow \underline{j}]$.

It follows that $\{\mathbf{NT}\} \vdash \chi_2 = \forall x((x = 0 \vee x = 1 \vee x = 2 \vee \ldots \vee x = n) \Rightarrow \psi)$. Having proved χ_1 and χ_2 from **NT**, we can deduce $\phi = \forall x((x < n) \Rightarrow \psi)$. □

It does not follow from Theorem 6.2 that for any bounded sentence ϕ, if $\mathbf{N} \not\models \phi$ then $\{\mathbf{NT}\} \vdash \neg\phi$. The reason is that the class of bounded sentences is *not* closed under negation! However, this class is still quite rich and useful. We shall see in the next section that, using bounded sentences, we can describe interesting properties of Turing machine computation. By Theorem 6.2, such properties will then be deducible from **NT**.

6.2 COMPUTATION AS A NUMBER-THEORETIC CONCEPT

In this section we shall show that the computation of Turing machines can be captured by certain expressions in number theory. Towards this goal, it will be useful to standardize our machines a little more. All of our Turing machines in this chapter have a single string, and halt at either "yes" or "no" (they may also fail to halt, of course). They never write a ▷ on their string (except, of course, when they see one), and so ▷ is always the unmistakable sign of the beginning of the string. The machine before halting moves all the way to the right, reaching the current end of its string (obviously, it must have a way of remembering the right end of the string, such as refraining from writing any ⊔'s), and then the state "yes" or "no" is entered. It should be clear that it is no loss of generality to assume that all Turing machines are in this standard form.

Consider now such a Turing machine $M = (K, \Sigma, \delta, s)$. We shall study in detail possible representations of computations and computational properties of M by numbers and expressions in number theory. The idea is the same as in Section 3.1: States and symbols in $K \cup \Sigma$ can be encoded as distinct integers. That is, we assume that $\Sigma = \{0, 1, \ldots, |\Sigma| - 1\}$, and $K = \{|\Sigma|, |\Sigma| + 1, \ldots, |\Sigma| + |K| - 1\}$. The starting state s is always encoded as $|\Sigma|$, and 0 always encodes ▷. "yes" and "no" are encoded as $\Sigma + 1$ and $\Sigma + 2$, respectively, and ⊔ is always encoded as a 1 (there is no need here to encode h, $\leftarrow$, $\rightarrow$, and $-$, as we did in Chapter 3). This requires $b = |\Sigma| + |K|$ integers *in toto*.

We can now encode *configurations* as sequences of integers in $\{0, 1, \ldots, b - 1\}$; equivalently, we can think of such sequences as integers in b-ary notation, with the most significant digit first. For example, configuration $C = (q, w, u)$, where $q \in K$ and $w, u \in \Sigma^*$ would be encoded by the sequence $C = (w_1, w_2, \ldots, w_m, q, u_1, u_2, \ldots, u_n)$, where $|w| = m$ and $|u| = n$, or equivalently by the unique integer whose b-ary representation is this sequence, namely $\sum_{i=1}^{m} w_i \cdot b^{m+n+1-i} + q \cdot b^n + \sum_{i=1}^{n} u_i \cdot b^{n-i}$.

Example 6.4: Let us consider the Turing machine $M = (K, \Sigma, \delta, s)$, with $K = \{s, q, \text{"yes"}, \text{"no"}\}$, $\Sigma = \{\triangleright, \sqcup, a, b\}$, and δ as shown in Figure 6.2; as always, we omit rules that will never be encountered in a legal computation.

$p \in K$,	$\sigma \in \Sigma$	$\delta(p, \sigma)$
s,	a	$(s, a, \rightarrow)$
s,	b	$(s, b, \rightarrow)$
s,	$\sqcup$	$(q, \sqcup, \leftarrow)$
s,	$\triangleright$	$(s, \triangleright, \rightarrow)$
q,	a	$(q, \sqcup, \leftarrow)$
q,	b	("no", b, $-$)
q,	$\triangleright$	("yes", $\triangleright$, $\rightarrow$)

Figure 6.2. A Turing machine.

M is a simple machine that checks, from right to left, whether all of its input symbols are a's, and if not rejects. It also erases the a's it finds by overwriting $\sqcup$'s. M deviates from our standardization in ways that are not important right now (e.g., it halts with the cursor in the middle of the string, as opposed to the right end). For this machine, $|K| = 4$ and $|\Sigma| = 4$, thus $b = 8$. We encode $\triangleright$ as 0, $\sqcup$ as 1, a as 2, b as 3, s as 4, "yes" as 5, "no" as 6, and q as 7. The configuration $(q, \triangleright aa, \sqcup\sqcup)$ is represented by the sequence $(0, 2, 2, 7, 1, 1)$, or equivalently by the integer $022711_8 = 9673_{10}$. Notice that, by our conventions, all configuration encodings begin with a 0, corresponding to $\triangleright$. Since we can assume that our Turing machines never write a $\triangleright$ (and thus no legal configuration starts with $\triangleright\triangleright$), the encoding is unique. □

Our main interest lies in the *yields* relation of a Turing machine. For example, $(q, \triangleright aa, \sqcup\sqcup) \overset{M}{\rightarrow} (q, \triangleright a, \sqcup \sqcup \sqcup)$ in the machine of Figure 6.2. Now, the "yields in one step" relation over configurations of M defines, via our encoding of configurations as integers in b-ary, a relation $Y_M \subseteq \mathbf{N}^2$ over the integers. Here is an intriguing possibility: Can we express this relation as an expression in number theory? Is there an expression $\text{yields}_M(x, y)$ in number theory with two free variables x and y such that $\mathbf{N}_{x=m,y=n} \models \text{yields}_M(x, y)$ if and only if $Y_M(m, n)$?

Let us consider two integers related by Y_M, for example, those corresponding to the configurations $(q, \triangleright aa, \sqcup\sqcup)$ and $(q, \triangleright a, \sqcup \sqcup \sqcup)$ above, $m = 022711_8$ and $n = 027111_8$. Obviously, $Y_M(m, n)$. The question is, how could we tell, just by looking at the two numbers? There is a simple way to tell: m and n are identical *except that the substring* 271_8 *in the* b*-ary expansion of* m *was replaced by* 711_8 *in the* b*-ary expansion of* n. Recalling what the digits stand for, this corresponds to the rule $\delta(q, a) = (q, \sqcup, \leftarrow)$ (fifth row of the table in Figure 6.2). Notice that this particular change could be described as a local replacement of the two digits 27_8 by 71_8; however, the third digit is useful when there is a move to the right. For uniformity, we think that a move of M entails

$$
\begin{array}{|l|}
\hline
042_8 \rightarrow 024_8 \\
043_8 \rightarrow 034_8 \\
041_8 \rightarrow 014_8 \\
242_8 \rightarrow 224_8 \\
243_8 \rightarrow 234_8 \\
241_8 \rightarrow 214_8 \\
342_8 \rightarrow 324_8 \\
343_8 \rightarrow 334_8 \\
341_8 \rightarrow 314_8 \\
141_8 \rightarrow 711_8 \\
271_8 \rightarrow 711_8 \\
071_8 \rightarrow 015_8 \\
371_8 \rightarrow 361_8 \\
\hline
\end{array}
$$

Figure 6.3. The table.

local replacement of triples of digits. *The complete table of these triples and their replacements. for the machine M of Figure 6.2, is shown in Figure 6.3.*

We can express the table in Figure 6.3 as an expression in number theory called, appropriately, $\text{table}_M(x, y)$, where M is the machine being simulated. In our example $\text{table}_M(x, y)$ is this expression:

$$((x = 042_8 \wedge y = 024_8) \vee (x = 043_8 \wedge y = 034_8) \vee \ldots \vee (x = 371_8 \wedge y = 361_8)).$$

Obviously, if two integers m and n satisfy this expression, then they must be the three-digit numbers representing a move by M.

Let us consider a sequence of octal integers that correspond to the computation of M on input aa: $0422_8, 0242_8, 0224_8, 02241_8, 02214_8, 02271_8, \ldots$. Notice that the fourth octal number is the third extended by a 1—a $\sqcup$. Such steps, deviating from our triple replacements, are necessary to capture the expansion of the string, when a blank symbol is encountered at the right end of the string. If we had not inserted this fourth number, we could not apply the triple substitution $241_8 \rightarrow 214_8$ to continue the computation. We say that 0224 was *padded* by a $\sqcup$ to obtain 02241. Only octal numbers whose last digit is a state (a digit $\geq |\Sigma|$) can be thus padded.

We are now ready to express the "yields" relation $Y_M(m, n)$ as an expression in number theory. This expression, $\text{yields}_M(x, x')$, is shown below. $\text{pads}_M(x, x')$ is an expression stating that the configuration represented by x has a state as its last component and x' is just x padded by a $\sqcup$ (we consider padding as a special case of yielding). Also, $\text{conf}_M(x)$ states that x encodes a configuration; both $\text{pads}_M(x, x')$ and $\text{conf}_M(x)$ are detailed later. The second

row of yields_M contains a sequence of bounded existential quantifiers, introducing several numbers bounded from above by x. The first such number, y, points to a digit in the b-ary expansion of x and x'. The number z_1 is the integer in b-ary spelled by the y last digits of x *and* x'. z_3 spells the next three digits of x (and z_3' of x'). Also, x and x' must agree in their digits other than their last $y+3$ ones (both numbers must spell z_4). Thus, x and x' must agree in all but three of their digits. Finally, the last row requires that the remaining three digits be related according to the table of $\bar{M}$'s moves.

$$\begin{aligned}&\text{yields}_M(x,x') = \underline{\text{pads}_M(x,x')\vee}\\&\underline{(\exists y<x)(\exists z_1<x)(\exists z_2<x)(\exists z_2'<x)(\exists z_3<x)(\exists z_3'<x)(\exists z_4<x)}\\&\underline{(\text{conf}_M(x)\wedge\text{conf}_M(x')\wedge}\\&\underline{\text{mod}(x,b\uparrow y,z_1)\wedge\text{div}(x,b\uparrow y,z_2)\wedge\text{mod}(x',b\uparrow y,z_1)\wedge\text{div}(x',b\uparrow y,z_2')\wedge}\\&\underline{\text{mod}(z_2,b\uparrow 3,z_3)\wedge\text{div}(z_2,b\uparrow 3,z_4)\wedge\text{mod}(z_2',b\uparrow 3,z_3')\wedge\text{div}(z_2',b\uparrow 3,z_4)\wedge}\\&\underline{\text{table}_M(z_3,z_3'))}.\end{aligned}$$

Here is a simple way to express $\text{pads}_M(x,x')$:

$$\underline{(\forall y<x)(\text{mod}(x,b,y)\Rightarrow y\geq|\Sigma|)\wedge x'=x\times b+1}.$$

Recall that $\text{conf}_M(x)$ states that the b-ary representation of x correctly encodes a configuration of M, that is, contains only one state digit, and no 0 digits:

$$\underline{(\exists y<x)(\forall z<x)(\text{state}_M(x,y)\wedge(\text{state}_M(x,z)\Rightarrow y=z)\wedge\text{nozeros}_M(x))},$$

where $\text{nozeros}_M(x)$ states that there are no zero digits in x in b-ary (except perhaps for the beginning, where they cannot be detected):

$$\underline{(\forall y<x)(\forall u<x)(\text{mod}(x,b\uparrow y,u)\wedge\text{mod}(x,b\uparrow(y+1),u)\Rightarrow x=u)},$$

and $\text{state}_M(x,y)$ expresses that the yth least significant b-ary digit of x encodes a state, that is, is a digit at least as large as $|\Sigma|$:

$$\text{state}_M(x,y) = \underline{(\exists z<x)(\exists w<x)(\text{div}(x,b\uparrow y,z)\wedge\text{mod}(z,b,w)\wedge|\Sigma|\leq w)}.$$

Finally, we can encode *whole computations* of M by juxtaposing consecutive configurations to form a single b-ary integer (followed by a 0, so that all configurations begin and end with a 0). The following is a way of stating that x encodes a halting computation of M starting from the empty string: Each configuration (substring of the b-ary expansion of x between two consecutive

0's) yields the next and, moreover, x starts with the three digits $0, |\Sigma|, 0$ (encoding $\triangleright s \triangleright$), and ends with the two digits $|\Sigma| + 1, 0$ (if the answer is "yes"), or $|\Sigma| + 2, 0$ ("no" answer). We can write this as an expression, called $\text{comp}_M(x)$:

$$(\forall y_1 < x)(\forall y_2 < x)(\forall y_3 < x)(\forall z_1 < x)(\forall z_2 < x)(\forall z_3 < x)(\forall u < x)(\forall u' < x)$$
$$((\text{mod}(x, b \uparrow y_1, z_1) \wedge \text{mod}(x, b \uparrow (y_1 + 1), z_1) \wedge \text{div}(x, b \uparrow (y_1 + 1), z_2) \wedge$$
$$\text{mod}(z_2, b \uparrow y_2, u) \wedge \text{mod}(z_2, b \uparrow (y_2 + 1), u) \wedge \text{div}(z_2, b \uparrow (y_2 + 1), z_3) \wedge$$
$$\text{mod}(z_3, b \uparrow y_3, u') \wedge \text{mod}(z_3, b \uparrow (y_3 + 1), u') \wedge$$
$$\text{conf}_M(u) \wedge \text{conf}_M(u')) \Rightarrow$$
$$\text{yields}_M(u', u))$$
$$\wedge(\forall u < x)(\text{mod}(x, b \uparrow 2, u) \Rightarrow (u = b \cdot (|\Sigma| + 1) \vee u = b \cdot (|\Sigma| + 2)))$$
$$\wedge(\forall u < x)(\forall y < x)((\text{div}(x, b \uparrow y, u) \wedge u < b \uparrow 2 \wedge b \leq u) \Rightarrow u = b \cdot |\Sigma|).$$

The second line states that the $y_1 + 1$st digit of x from the right is a 0, and likewise the third line for the $y_2 + 1$st digit and the fourth for the $y_3 + 1$st. The fifth line claims that u and u', the numbers spelled by the digits in between, are configurations and thus contain no zeros. When these conditions are satisfied, the sixth line demands that u yield u'. The seventh line states that the last two digits are $|\Sigma| + 1, 0$ or $|\Sigma| + 2, 0$ (that is, the machine halts), while the last line says that the first three digits are $0, |\Sigma|, 0$ spelling $\triangleright s \triangleright$ (that is, the machine starts with an empty string).

The preceding discussion of the expression comp_M and its constituents establishes the main result of this section:

Lemma 6.1: For each Turing machine M we can construct a bounded expression $\text{comp}_M(x)$ in number theory satisfying the following: For all nonnegative integers n we have that $\mathbf{N}_{x=n} \models \text{comp}_M(x)$ if and only if the b-ary expansion of n is the juxtaposition of consecutive configurations of a halting computation of M, starting from the empty string. □

6.3 UNDECIDABILITY AND INCOMPLETENESS

The stage has been set for a remarkable result. Figure 6.4 depicts several interesting sets of sentences in the vocabulary of number theory. From left to right, we first have the set of sentences that are valid, satisfied by all models. We know that this set of sentences is recursively enumerable, because by the completeness theorem it coincides with the set of all sentences with validity proofs. The next set contains all properties of the integers that can be proven from NT (clearly, this includes the valid sentences, which can be proved without resorting to NT). A further superset is the set of all properties of the integers, all sentences satisfied by model $\mathbf{N}$. This is an important set of sentences, in

a certain way our motivation for studying logic. Finally, a further superset is the set of sentences that have *some* model, that are not *unsatisfiable* or, equivalently, *inconsistent*. As usual in such diagrams, negation corresponds to "mirroring" around the vertical axis of symmetry.

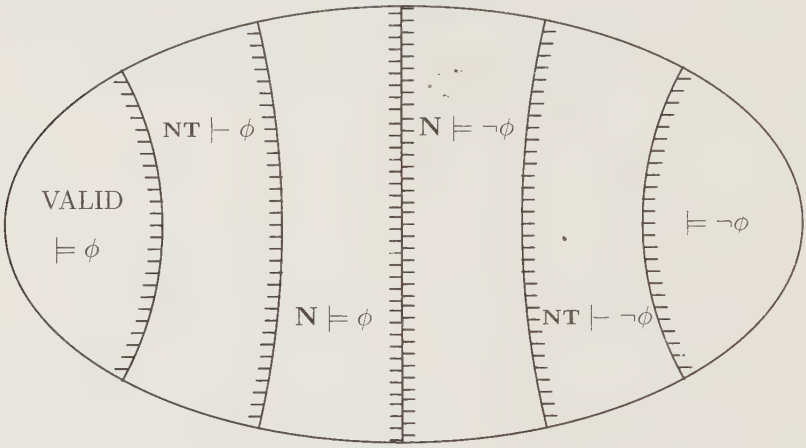


Figure 6-4. All sentences of number theory.

We next prove that the set of inconsistent sentences and the set of sentences provable from **NT** are *recursively inseparable*. That is, there is no recursive set that separates one from the other. (recall Section 3.3). This result has a host of important implications: *None of the five "boundaries" shown in Figure 6.4 can be recursive!*

Theorem 6.3: The set of unsatisfiable sentences and the set of sentences provable from **NT** are recursively inseparable.

Proof: We shall use the recursive inseparability of the languages $L_1' = \{M : M(\epsilon) = \text{"yes"}\}$ and $L_2' = \{M : M(\epsilon) = \text{"no"}\}$ (recall the Corollary to Theorem 3.4). Given any description M of a Turing machine, we shall show how to construct a sentence ϕ_M, such that: (a) If $M(\epsilon) =$ "yes", then $\{\mathbf{NT}\} \vdash \phi_M$; and (b) If If $M(\epsilon) =$ "no", then ϕ_M is unsatisfiable. This would prove the theorem, because, if there were an algorithm that separates unsatisfiable sentences from true properties of the integers, then we could use this algorithm, in conjunction with the algorithm for constructing ϕ_M, to separate Turing machines that accept the empty string from those that reject it, contradicting that result.

ϕ_M is simply the sentence $\underline{\mathbf{NT} \wedge \psi}$, where:

$$\psi = \underline{\exists x(\text{comp}_M(x) \wedge ((\forall y < x)\neg\text{comp}_M(y)) \wedge \text{mod}(x, b^2, b \cdot (|\Sigma| + 1)))}.$$

Intuitively, ψ states that there is an integer which is the smallest integer encoding a computation of M on the empty string, and furthermore the computation is accepting.

Suppose that $M(\epsilon) =$ "yes". Then clearly there is a unique computation of M that starts with the empty string, and finally halts at the "yes" state. That is, there is a unique integer n (probably unimaginably large), such that $\mathbf{N} \models \mathrm{comp}_M[x \leftarrow \underline{n}]$. Therefore, $\mathbf{N} \models \underline{\exists x \mathrm{comp}_M(x)}$, and furthermore, since n is unique, $\mathbf{N} \models \underline{\exists x \mathrm{comp}_M(x) \wedge ((\forall y < x) \neg \mathrm{comp}_M(y))}$. Finally, since the last two digits of the b-ary expansion of n are $|\Sigma| + 1$ and 0, $\mathbf{N} \models \psi$.

Now, ψ contains quantifiers that are bounded (except for the outermost existential one, recall the construction of $\mathrm{comp}_M(x)$ in the previous section), and hence it can be written as a bounded sentence in prenex normal form. It follows from Theorem 6.2 that $\{\mathbf{NT}\} \vdash \psi$, and hence certainly $\{\mathbf{NT}\} \vdash \phi_M$. So, we conclude that $M(\epsilon) =$ "yes" implies $\{\mathbf{NT}\} \vdash \phi_M$.

Suppose now that $M(\epsilon) =$ "no". We shall show that ϕ_M is inconsistent. Since $M(\epsilon) =$ "no", arguing as above we can show that $\mathbf{N} \models \phi'_M$, where

$$\phi'_M = \underline{\exists x' (\mathrm{comp}_M(x') \wedge ((\forall y < x') \neg \mathrm{comp}_M(y)) \wedge \mathrm{mod}(x', b^2, b \cdot (|\Sigma| + 2)))},$$

where we have omitted **NT**, and replaced $|\Sigma + 1|$ by $|\Sigma + 2|$ (the encoding of "no").

As before, it is easy to check that ϕ'_M can be rewritten as a bounded sentence, and thus $\mathbf{NT} \vdash \phi'_M$. We shall show that ϕ_M and ϕ'_M are inconsistent.

This is not so hard to see: ϕ_M asserts that x is the smallest integer such that $\mathrm{comp}_M(x)$, and its last digit is $|\Sigma + 1|$, while ϕ'_M asserts that x' is the smallest integer such that $\mathrm{comp}_M(x')$, and its last digit is $|\Sigma + 2|$. And we can prove from **NT** (which is included in ϕ_M) that this is impossible. Here is how this proof goes: We know (axiom **NT12**) that either $\underline{x < x'}$, or $\underline{x' < x}$, or $\underline{x = x'}$. Since $\mathrm{comp}_M(x)$ and $\underline{(\forall y < x') \neg \mathrm{comp}_M(y)}$, the first eventuality is excluded, and symmetrically for the second. Hence $x = x'$. But **NT14** asserts that mod is a function. We arrive at $b \cdot (|\Sigma| + 1) = b \cdot (|\Sigma| + 2)$, which contradicts axioms **NT10** and **NT13**. Hence, if $M(\epsilon) =$ "no", ϕ_M is inconsistent. The proof is complete. □

Corollary 1: The following problems are undecidable, given a sentence ϕ:

(a) VALIDITY, that is, is ϕ valid? (And, by the completeness theorem, so is THEOREMHOOD.)

(b) Does $\mathbf{N} \models \phi$?

(c) Does $\{\mathbf{NT}\} \vdash \phi$?

Proof: Each of these sets of sentences would separate the two sets shown recursively inseparable in the theorem. □

What is worse, Theorem 6.3 frustrates our ambition to axiomatize number theory:

Corollary 2 (Gödel's Incompleteness Theorem): There is no recursively enumerable set of axioms Ξ such that, for all expressions ϕ, $\Xi \vdash \phi$ if and only if $\mathbf{N} \models \phi$.

Proof: We first notice that, since Ξ is recursively enumerable, the set of all possible proofs from Ξ is also recursively enumerable: Given an alleged proof, the Turing machine that accepts this language checks for each expression in the sequence whether it is a logical axiom, or whether it follows by *modus ponens*, or whether it is in Ξ. The latter test invokes the machine that accepts Ξ. If all expressions in the sequence qualify then the proof is accepted, otherwise the machine diverges.

Since the set of all proofs from Ξ is recursively enumerable, there is a Turing machine that enumerates it (recall the definition just before Proposition 3.5), and hence there is a Turing machine that enumerates the set $\{\phi : \Xi \vdash \phi\}$, which is by hypothesis equal to $\{\phi : \mathbf{N} \models \phi\}$. Hence this latter set (the left-hand half of Figure 6.4) is recursively enumerable. Similarly, $\{\phi : \mathbf{N} \models \neg\phi\}$ is recursively enumerable (this is the right-hand half). Since these sets are complements of one another, we conclude by Proposition 3.4 that they are recursive, contradicting the previous corollary. □

Notice the strength of this statement: There can be no set of axioms that captures the true properties of the integers, not even an infinite one, not even a non-recursive one—as long as it is recursively enumerable. Any such sound axiomatic system must be *incomplete*, and there must exist a true property of the integers that is not provable by it.

Corollary 2 also establishes that $\{\phi : \mathbf{N} \models \phi\}$ is not recursively enumerable (because, if it were, it would certainly comprise a complete axiomatization of itself). For the same reason, neither is its complement recursively enumerable. Hence, the set of the true properties of the whole numbers, which we had hoped to understand and conquer, is a specimen from the vast upper region of Figure 3.1, an example of a set that is neither recursively enumerable, nor the complement of a recursively enumerable set.

6.4 NOTES, REFERENCES, AND PROBLEMS

6.4.1 David Hilbert, one of the greatest mathematicians of this century, visualized that mathematical proofs could one day be mechanized and automated; his main motivation was to mechanically verify the consistency of the axiomatic systems in use by mathematicians. That Hilbert's project seems to us today almost silly in its ambitious optimism is only evidence of the tremendous influence it has had on mathematical thought and culture. Some colossal early efforts at systematizing mathematics

- A. N. Whitehead, B. Russell *Principia Mathematica*, Cambridge Univ. Press, 1913,

as well as several important positive results for first-order logic (including not only algorithms for special cases, see Section 20.1 and Problem 20.2.11, but also the completeness theorem by Hilbert's own student Kurt Gödel, recall Theorem 5.7) can be seen as part of this major intellectual effort. On the other hand, the development of the theory of computation by Turing and others (recall the references in 2.8.1) was also motivated by this project, albeit in a negative way (hence the frequent references to the *Entscheidungsproblem*—"the decision problem"—in these papers).

The incompleteness theorem with its devastating consequences (Theorem 6.3 and its corollaries here), also by Gödel

- K. Gödel "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme" ("On formally undecidable theorems in Principia Mathematica and related systems), *Monatshefte für Mathematik und Physik, 38*, pp. 173–198, 1931,

put an abrupt end to Hilbert's project. In fact, Gödel proved incompleteness for number theory *without the exponentiation function* $\uparrow$. (Recall that exponentiation was at the basis of our technique of expressing computations as number-theoretic statements; in its absence a much more elaborate and indirect encoding technique must be employed, see for example the exposition in

- M. Machtey and P.R. Young *An Introduction to the General Theory of Algorithms*, Elsevier, New York, 1978.)

Also, Gödel's original proof of the incompleteness theorem was purely logical, with no recourse to computation, and so it did not immediately imply the undecidability of validity in first-order logic, which had to wait for Church and Turing's work (see the papers mentioned in Chapter 2).

6.4.2 As a fairly special case exemplifying his ambitious project, Hilbert asked in 1900 whether there is an algorithm that decides if a multivariate polynomial equation such as $x^2y + 3yz - y^2 - 17 = 0$ has an integer solution, fully expecting a positive answer. This problem has been known as *Hilbert's tenth problem*, because it was one of several fundamental mathematical problems proposed by Hilbert in

- D. Hilbert "Mathematical problems," *Bull. Amer. Math. Soc.*, 8, pp. 437–479, 1902. Lecture delivered at the World Congress of Mathematicians in Paris, 1900.

Hilbert's tenth problem is a very special case of the problem of telling whether $\mathbf{N} \models \phi$ for a sentence ϕ (in particular, ϕ is restricted to have no Boolean connectives, no

exponentiation, no universal quantifiers, and no inequality). But even this special case is undecidable:

- Y. Matiyasevich "Enumerable sets are Diophantine," *Dokl. Akad. Nauk SSSR, 191*, pp. 279–282; translation in *Soviet Math. Doklady, 11*, pp. 354–357, 1970;

for a nice exposition of this result see

- M. Davis "Hilbert's tenth problem is unsolvable," *American Math. Monthly, 80*, pp. 233 -269, 1973.

But other restrictions of number theory (basically, the case without exponentiation *or multiplication*, known as *Presburger Arithmetic*, see Problem 20.2.12), as well as the theory of real numbers (see Problem 20.2.12), are decidable.

6.4.3 Problem: Gödel's incompleteness theorem is based on the fact that, if number theory were axiomatizable, then it would be decidable. In Example 5.5 we noticed that group theory is definitely axiomatizable (recall the axioms **GR1** through **GR3**); however, group theory is *not* decidable; this latter result is from

- A. Tarski, A. Mostowski, and R. M. Robinson *Undecidable Theories*, North-Holland, Amsterdam, 1953.

What is the basic difference between group theory (the sentences true of all groups) and number theory (the sentences true of $\mathbf{N}$) that explains this discrepancy?

PART

III P AND NP

"The classes of problems which are respectively known and not known to have good algorithms are of great theoretical interest. [...] I conjecture that there is no good algorithm for the traveling salesman problem. My reasons are the same as for any mathematical conjecture: (1) It is a legitimate mathematical possibility, and (2) I do not know."

Jack Edmonds, 1966

CHAPTER 7

RELATIONS BETWEEN COMPLEXITY CLASSES

This chapter recounts all we know about inclusion relations between complexity classes. Unfortunately, what we know does not amount to much...

7.1 COMPLEXITY CLASSES

We have already seen the concept of a complexity class while developing our understanding of Turing machines and their capabilities. We shall now study complexity classes in earnest.

A complexity class is specified by several parameters. First, the underlying *model of computation*, which we have already decided to fix as the *multi-string* Turing machine—a convention which, we have argued, does not matter much. Second, a complexity class is characterized by a *mode of computation* (basically, when do we think that a machine accepts its input). We have already seen the two most important modes, namely the *deterministic mode* and the *nondeterministic mode*. More will come (see Problem 20.2.14 for a general and comprehensive treatment of modes). Thirdly, we must fix a *resource* we wish to bound —something expensive the machine uses up. We have seen the basic resources *time* and *space*, but there are many more (see for example Problem 7.4.12). Finally, we must specify a *bound*, that is, a function f mapping nonnegative integers to nonnegative integers. The complexity class is then defined as the set of all languages decided by some multi-string Turing machine M operating in the appropriate mode, and such that, for any input x, M expends at most $f(|x|)$ units of the specified resource.

What kinds of bounding functions f should we consider? In principle, any function from the nonnegative integers to the nonnegative integers can be used

to define a complexity class; this includes "weird" functions such as the following function $f(n)$: "if the n is a prime then, then $f(n)$ is 2^n, otherwise it is the nth smallest number such that... etc." We could even have a complexity function which is so complicated that it *cannot itself be computed* within the time or space it allows! The use of such functions opens up possibilities that can be very intricate and confusing (see Theorem 7.3), and still quite uninteresting with respect to understanding the complexity of natural computational problems.

Definition 7.1: We shall now define the broad and natural class of functions that we shall use as bounds in this book. Let f be a function from the nonnegative integers to the nonnegative integers. We say that f is a *proper complexity function* if f is nondecreasing (that is, $f(n+1) \geq f(n)$ for all n), and furthermore the following is true: There is a k-string Turing machine $M_f = (K, \Sigma, \delta, s)$ *with input and output* such that, for any integer n, and any input x of length n, $(s, \triangleright, x, \triangleright, \epsilon, \ldots, \triangleright, \epsilon) \overset{M_f}{\overset{t}{\rightarrow}} (h, x, \triangleright, \sqcup^{j_2}, \triangleright, \sqcup^{j_3}, \ldots, \triangleright, \sqcup^{j_{k-1}}, \triangleright, \sqcap^{f(|x|)})$, such that $t = \mathcal{O}(n + f(n))$, and the $j_i = \mathcal{O}(f(|x|))$ for $i = 2, \ldots, k-1$, with t and the j_i's depending only on n. In other words, on input x, M_f computes the string $\sqcap^{f(|x|)}$, where $\sqcap$ is a "quasi-blank" symbol. And, on any input x, M_f halts after $\mathcal{O}(|x| + f(|x|))$ steps and uses $\mathcal{O}(f(|x|))$ space besides its input. □

Example 7.1: The function $f(n) = c$, where c a fixed integer, is definitely a proper complexity function: M_f just writes a $\sqcap^c$ on its last string, ignoring the input. So is the function $f(n) = n$: M_f rewrites all input symbols as quasiblanks in the last string. The function $f(n) = \lceil \log n \rceil$ is also proper: M_f is the following three-string machine: Its first cursor moves slowly from left to right, while the second string *counts in binary* the number of input symbols (say, using the binary successor Turing machine from Example 2.2 as a subroutine). When it sees the first blank on its input, its second has length $f(n) = \lceil \log n \rceil$. M_f has just to erase the second string, copying all symbols to the output as $\sqcap$'s. The time required is $\mathcal{O}(|x|)$. Naturally, all these functions are also nondecreasing, as required by the definition.

It turns out that the class of proper complexity functions is very extensive. It may exclude many "pathological" functions, but it does include essentially all "reasonable" functions one would expect to use in the analysis algorithms and the study of their complexity. For example, all of these functions are proper: $\log n^2$, $n \log n$, n^2, $n^3 + 3n$, 2^n, $\sqrt{n}$, and $n!$. To see why, it is quite easy to prove that, if functions f and g are proper, then so are $f+g$, $f \cdot g$, and 2^g, among others (the machine for the new function first computes the constituent functions, and then computes the new function by appropriate simple motions of its cursors, see Problem 7.4.3). For $\sqrt{n}$ and $n!$ other arguments work (Problem 7.4.3). □

We shall henceforth use only proper complexity functions in relation to complexity classes. This leads to much convenience and standardization, as we explain next. Let us say that a Turing machine M (with input and output

or not, deterministic or nondeterministic) is *precise* if, there are functions f and g such that for every $n \geq 0$, for every input x of length n, and for every computation of M (in the case that M is nondeterministic), M halts after precisely $f(n)$ steps, and all of its strings, except for the first and last (in the case that M is a machine with input and output), are at halting of length precisely $g(n)$.

Proposition 7.1: Suppose that a (deterministic or nondeterministic) Turing machine M decides a language L within time (or space) $f(n)$, where f is a proper function. Then there is a precise Turing machine M', which decides the same language in time (or space, respectively) $\mathcal{O}(f(n))$.

Proof: In all four cases (deterministic time, deterministic space, nondeterministic time, or nondeterministic space) the machine M' on input x starts off by simulating the machine M_f associated with the proper function f on x, using a new set of strings. After M_f's computation has halted, M uses the output string of M_f as a "yardstick" of length $f(|x|)$, to guide its further computation. If $f(n)$ is a time bound, then M' simulates M on a different set of strings, *using the yardstick as an "alarm clock"*. That is, it advances its cursor on the yardstick after the simulation of each step of M, and halts if and only if a true blank is encountered, after precisely $f(|x|)$ steps of M. If $f(n)$ is a space bound, then M' simulates M on the quasiblanks of M_f's output string. In either case, the machine is precise. If M is nondeterministic, then precisely the same amount of time or space is used over all possible computations of M' on x, as this amount depends only on the deterministic phase of M' (the simulation of M_f). In both cases, the time or space is the time or space consumed by M_f, plus that consumed by M, and is therefore $\mathcal{O}(f(n))$.

In the case of space-bounded machines, the "space yardstick" can be used by M' to also "count" steps in an appropriate base representing the size of the alphabet of M, and thus M' will never indulge in a meaninglessly long computation. Thus, we can assume that space-bounded machines always halt. □

We shall henceforth consider complexity classes of the form **TIME**(f) (deterministic time), **SPACE**(f) (deterministic space), **NTIME**(f) (nondeterministic time) and **NSPACE**(f) (nondeterministic space). In all four cases, function f will always be a proper complexity function (with the single exception of Theorem 7.3, whose purpose is precisely to illustrate the kinds of absurdities that result in the absence of this convention).

Sometimes we shall use in the place of f not a particular function, but a *family* of functions, parametrized by an integer $k > 0$. The complexity class denoted is the *union* of all individual complexity classes, one for each value of k. Two most important examples of such parametrized complexity classes are

these:

$$\mathbf{TIME}(n^k) = \bigcup_{j>0} \mathbf{TIME}(n^j)$$

and its nondeterministic counterpart

$$\mathbf{NTIME}(n^k) = \bigcup_{j>0} \mathbf{NTIME}(n^j),$$

which we already know as **P** and **NP**, respectively. Other important complexity class of this sort are $\mathbf{PSPACE} = \mathbf{SPACE}(n^k)$, $\mathbf{NPSPACE} = \mathbf{NSPACE}(n^k)$, and $\mathbf{EXP} = \mathbf{TIME}(2^{n^k})$. Finally, for space classes we can look at sublinear bounds; two important classes here are $\mathbf{L} = \mathbf{SPACE}(\log n)$ and $\mathbf{NL} = \mathbf{NSPACE}(\log n)$.

Complements of Nondeterministic Classes

When we first defined nondeterministic computation in Section 2.7, we were struck by an asymmetry in the way "yes" and "no" inputs are treated. For a string to be established as a string in the language (a "yes" input), one successful computational path is enough. In contrast, for a string not in the language, all computational paths must be unsuccessful. As a result of a very similar asymmetry, the two classes **RE** (all recursively enumerable languages) and **coRE** (the complements of such languages) were shown to be different in Section 3.3 (for example, the "halting" language H is in $\mathbf{RE} - \mathbf{coRE}$; recall Figure 3.1).

Let $L \subseteq \Sigma^*$ be a language. The *complement* of L, denoted $\overline{L}$, is the language $\Sigma^* - L$, that is, the set of all strings in the appropriate alphabet that are not in L. We now extend this definition to decision problems. The complement of a decision problem A, usually called A COMPLEMENT, will be defined as the decision problem whose answer is "yes" whenever the input is a "no" input of A, and vice versa. For example, SAT COMPLEMENT is this problem: Given a Boolean expression ϕ in conjunctive normal form, is it *unsatisfiable?* HAMILTON PATH COMPLEMENT is the following: Given a graph G, is it true that G *does not* have a Hamilton path? And so on. Notice that, strictly speaking, the languages corresponding to the problems HAMILTON PATH and HAMILTON PATH COMPLEMENT, for example, are not complements of one another, as their union is not Σ^* but rather the set of all strings that encode graphs; this is of course another convenient convention with no deep consequences.

For any complexity class $\mathcal{C}$, $\mathbf{co}\mathcal{C}$ denotes the class $\{\overline{L} : L \in \mathcal{C}\}$. It is immediately obvious that if $\mathcal{C}$ is a deterministic time or space complexity class, then $\mathcal{C} = \mathbf{co}\mathcal{C}$; that is, *all deterministic time and space complexity classes are closed under complement.* The reason is that any deterministic Turing machine deciding L within a time or space bound can be converted to a deterministic

Turing machine that decides $\overline{L}$ within the same time or space bound: The same machine, with the roles of "yes" and "no" reversed. As we shall see in Section 7.3, a more elaborate argument establishes the same for nondeterministic *space* classes. *But it is an important open problem whether nondeterministic time complexity classes are closed under complement.*

7.2 THE HIERARCHY THEOREM

The theory of computation has concerned itself with hierarchies from its inception (see Problem 3.4.2 for the *Chomsky* hierarchy of recursively enumerable languages). Many results state that the addition of a new feature (a push-down store, nondeterminism, the power to overwrite the input, etc.) enhances a model's computational capabilities. We have already seen the most classical and fundamental of these results: In Chapter 3 we have shown that the recursive languages constitute a proper subset of the recursively enumerable ones. In other words, the ability of a Turing machine to reject its input by simply diverging results in a richer class of languages, one containing the non-recursive language H.

In this section we prove a *quantitative* hierarchy result: With sufficiently greater time allocation, Turing machines are able to perform more complex computational tasks. Predictably, our proof will employ a quantitative sort of diagonalization.

Let $f(n) \geq n$ be a proper complexity function, and define H_f to be the following time-bounded version of the HALTING language H:

$$H_f = \{M; x : M \text{ accepts input } x \text{ after at most } f(|x|) \text{ steps}\},$$

where M in the definition of H_f ranges over all descriptions of deterministic multi-string Turing machines (in our standard description of Turing machines, recall Section 3.1). We shall assume that, although a machine may have an arbitrary alphabet (so that we can use Theorem 2.2), the languages of interest, and therefore the inputs x, contain the symbols used for encoding Turing machines (0, 1, "(" ";" etc., recall Section 3.1). This is no loss of generality, since languages over different alphabets obviously have identical complexity properties, as long as these alphabets contain at least two symbols. Thus, input x is not encoded, but presented "verbatim."

This following result is now the analog of Proposition 3.1:

Lemma 7.1: $H_f \in \mathbf{TIME}((f(n))^3)$.

Proof: We shall describe a Turing machine U_f with four strings, which decides H_f in time $(f(n))^3$. U_f is based on several machines that we have seen previously: The universal Turing machine U described in Section 3.1; the single-string simulator of multi-string machines (Theorem 2.1); the linear speedup

machine that shaves constants off the time bound (Theorem 2.2); and the machine M_f that computes a "yardstick" of length precisely $f(n)$, which exists because we are assuming that f is proper. The synthesis of all these simple ideas presents no conceptual problems, but some care is needed in the details.

First U_f uses M_f on the second part of its input, x, to initialize on its fourth string an appropriate "alarm clock" $\sqcap^{f(|x|)}$, to be used during the simulation of M. (Here we assume that M_f has at most four strings; if not, the number of strings of U_f has to be increased accordingly.) M_f operates within time $\mathcal{O}(f(|x|))$ (where the constant depends on f alone, and not on x or M). U_f also copies the description of the machine M to be simulated on its third string, and converts x on its first string to the encoding of $\triangleright x$. The second string is initialized to the encoding of the initial state s. We can also at this point check that indeed M is the description of a legitimate Turing machine, and reject if it is not (this requires linear time using two strings). The total time required so far is $\mathcal{O}(f(|x|)+n) = \mathcal{O}(f(n))$.

The main operation of U_f starts after this initial stage. Like U in Section 3.1, U_f simulates one-by-one the steps of M on input x. Exactly as in the proof of Theorem 2.1, the simulation is confined in the first string, where the encodings of the contents of all strings of M are kept. Each step of M is simulated by two successive scans of the first string of U_f. During the first scan, U_f collects all relevant information concerning the currently scanned symbols of M, and writes this information on the second string. The second string also contains the encoding of the current state. U_f then matches the contents of the second string with those of the third (containing the description of M) to find the appropriate transition of M. U_f goes on to perform the appropriate changes to its first string, in a second pass. It also advances its "alarm clock" by one.

U_f simulates each step of M in time $\mathcal{O}(\ell_M k_M^2 f(|x|))$, where k_M is the number of strings of M, and ℓ_M is the length of the description of each state and symbol of M. Since, for legitimate Turing machines, these quantities are bounded above by the logarithm of the length of the description of the machine, the time to simulate each step of M is $\mathcal{O}(f^2(n))$, where, once more, the constant does not depend on M.

If U_f finds that M indeed accepts x within $f(|x|)$ steps, it accepts its input $M;x$. If not (that is, if M rejects x, or if the alarm clock expires) then U_f rejects its input. The total time is $\mathcal{O}(f(n)^3)$. It can be easily made at most $f(n)^3$, by modifying U_f to treat several symbols as one, as in the proof of the linear speedup theorem (Theorem 2.1). □

The next result now is the time-bounded analog of Theorem 3.1:

Lemma 7.2: $H_f \notin \mathbf{TIME}(f(\lfloor \frac{n}{2} \rfloor))$.

Proof: Suppose, for the sake of contradiction, that there is a Turing machine

M_{H_f} that decides H_f in time $f(\lfloor \frac{n}{2} \rfloor)$. This leads to the construction of a "diagonalizing" machine D_f, with the following behavior:

$$D_f(M) : \textbf{if } M_{H_f}(M; M) = \text{"yes"} \textbf{ then "no" else "yes"}$$

D_f on input M runs in the same time as M_{H_f} on input $M; M$, that is, in time $f(\lfloor \frac{2n+1}{2} \rfloor) = f(n)$.

Does then D_f accept its own description? Suppose that $D_f(D_f) =$ "yes". This means that $M_{H_f}(D_f; D_f) =$ "no", or, equivalently, $D_f; D_f \notin H_f$. By the definition of H_f, this means that D_f fails to accept its own description in $f(n)$ steps, and, since we know that D_f always accepts or rejects its input within $f(n)$ steps, $D_f(D_f) =$ "no". Similarly, $D_f(D_f) =$ "no" implies $D_f(D_f) =$ "yes", and our assumption that $H_f \in \textbf{TIME}(f(\lfloor \frac{n}{2} \rfloor))$ has led us to a contradiction. □

Comparing lemmata 7.1 and 7.2 we have:

Theorem 7.1 (The Time Hierarchy Theorem): If $f(n) \geq n$ is a proper complexity function, then the class $\textbf{TIME}(f(n))$ is strictly contained within $\textbf{TIME}((f(2n+1))^3)$. □

In fact, the time hierarchy is much denser than what Theorem 7.1 indicates, as $(f(2n+1))^3$ in the statement of Theorem 7.1 can be replaced by much slower-growing functions such as $f(n)\log^2 f(n)$ (see Problem 7.4.8). For us, the important attribute of function $(f(2n+1))^3$ in the above theorem is that *it is a polynomial whenever $f(n)$ is a polynomial;* this yields the following result:

Corollary: **P** is a proper subset of **EXP**.

Proof: Any polynomial will ultimately become smaller than 2^n, and so **P** is a subset of $\textbf{TIME}(2^n)$ (and thus of **EXP**). To prove proper inclusion, notice that, by Theorem 7.1, $\textbf{TIME}(2^n)$ (which contains all of **P**) is a proper subset of $\textbf{TIME}((2^{2n+1})^3) \subseteq \textbf{TIME}(2^{n^2})$, which is a subset of **EXP**. □

For space, we can prove the following result, which follows from two lemmata very similar to lemmata 7.1 and 7.2 (see Problem 7.4.9):

Theorem 7.2 (The Space Hierarchy Theorem): If $f(n)$ is a proper function, then $\textbf{SPACE}(f(n))$ is a proper subset of $\textbf{SPACE}(f(n)\log f(n))$. □

We shall end this section by proving a result that superficially seems to contradict Theorem 7.1. In fact, this result is a warning that, in the presence of complexity functions that are not proper, very counterintuitive phenomena can take place:

Theorem 7.3 (The Gap Theorem): There is a recursive function f from the nonnegative integers to the nonnegative integers such that $\textbf{TIME}(f(n)) = \textbf{TIME}(2^{f(n)})$.

Proof: We shall define f in such a way that no Turing machine, computing on an input of length n, halts after a number of steps between $f(n)$ and $2^{f(n)}$.

That is, for an input of length n, a machine M will either halt before $f(n)$ steps, or halt after $2^{f(n)}$ steps (or it will not halt at all).

We consider all Turing machines, say in the lexicographic order of their encoding, $M_0, M_1, M_2, \ldots$ A machine on this list may fail to halt on some or all inputs. For $i, k \geq 0$, we define the following property $P(i,k)$: "Any machine among $M_0, M_1, \ldots, M_i$, on any input of length i, will either halt after fewer than k steps, or it will halt after more than 2^k steps, or it will not halt at all." Even though a machine may fail to halt on many inputs, $P(i,k)$ can be decided by simulating all machines of index i or less on all inputs of length i for up to $2^k + 1$ steps.

Let us now define the value $f(i)$ for some $i \geq 0$. Consider the following sequence of values of k: $k_1 = 2i$, and, for $j = 2, 3, \ldots$, $k_j = 2^{k_{j-1}} + 1$. Let $N(i) = \sum_{j=0}^{i} |\Sigma_j|^i$, that is, the total number of inputs of length i to the first $i+1$ machines. Since each such input can make $P(i,k_j)$ false for at most one value of k_j, there must be an integer $\ell \leq N(i)$ such that $P(i,k_\ell)$ is true. *We are now ready to define* $f(i)$: $f(i) = k_\ell$ (notice the fantastically fast growth, as well as the decidedly unnatural definition of this function).

Consider now a language L in $\mathbf{TIME}(2^{f(n)})$. L is decided by some Turing machine, call it M_j, within time $2^{f(n)}$. Then, for any input x with $|x| \geq j$ (that is, for all inputs except for finitely many) it is impossible that M_j halts after a number of steps that lies between $f(|x|)$ and $2^{f(|x|)}$ (just because the values of $f(n)$ with $n \geq j$ have been so designed, with M_j taken into account). Since M_j definitely halts after at most $2^{f(|x|)}$ steps, we must conclude that it halts after at most $f(|x|)$ steps. Of course, there are the finitely many inputs (those of length less than j) for which we do not know when M_j halts. But we can modify M_j, augmenting its set of states accordingly, so that the resulting machine M_j' indeed decides all these finitely many inputs in time less than twice the length of the input. It follows that $L \in \mathbf{TIME}(f(n))$, and thus $\mathbf{TIME}(f(n)) = \mathbf{TIME}(2^{f(n)})$. □

It is easy to see that 2^n in this statement can be replaced by any fast-growing recursive function; also, a similar result is true for space complexity (Problem 7.4.11).

7.3 THE REACHABILITY METHOD

The hierarchy theorems tell us how classes of the same kind (deterministic time, deterministic space) relate to each other when we vary the function that represents the complexity bound; they were the very first facts proven about complexity. Similar results, although much harder to prove, are known for nondeterministic complexity classes (see Problem 7.4.10). However, the most interesting, persistent, and perplexing questions in complexity theory concern the relationship between classes *of different kinds*—such as **P** versus **NP**. In

this section we shall prove the very few results of this sort that we know.

Theorem 7.4: Suppose that $f(n)$ is a proper complexity function. Then:
(a) $\mathbf{SPACE}(f(n)) \subseteq \mathbf{NSPACE}(f(n))$ and $\mathbf{TIME}(f(n)) \subseteq \mathbf{NTIME}(f(n))$.
(b) $\mathbf{NTIME}(f(n)) \subseteq \mathbf{SPACE}((f(n))$.
(c) $\mathbf{NSPACE}(f(n)) \subseteq \mathbf{TIME}(k^{\log n + f(n)})$.

Proof: Part (a) is trivial: Any deterministic Turing machine is also a nondeterministic one (with just one choice for each step), and so any language in $\mathbf{SPACE}(f(n))$ is also in $\mathbf{NSPACE}(f(n))$, and similarly for $\mathbf{TIME}(f(n))$ and $\mathbf{NTIME}(f(n))$.

To show (b), consider a language $L \in \mathbf{NTIME}(f(n))$. There is a precise nondeterministic Turing machine M that decides L in time $f(n)$. We shall design a deterministic machine M' that decides L in *space* $f(n)$.

The idea is a simple one, and familiar from Theorem 2.6 where a nondeterministic machine was simulated in exponential time (so Theorem 2.6 follows from parts (b) and (c) of the present one). The deterministic machine M' generates a sequence of nondeterministic choices for M, that is, an $f(n)$-long sequence of integers between 0 and $d-1$ (where d is the maximum number of choices for any state-symbol combination of M). Then M' simulates the operation of M with the given choices. This simulation can obviously be carried out in space $f(n)$ (in time $f(n)$, only $\mathcal{O}(f(n))$ characters can be written!). However, there are exponentially many such simulations that must be tried, to check whether a sequence of choices that leads to acceptance exists. The point is that *they can be carried out one-by-one, always erasing the previous simulation to reuse space.* We only need to keep track of the sequence of choices currently simulated, and generate the next, but both tasks can easily be done within space $\mathcal{O}(f(n))$. The fact that f is proper can be used to generate the first sequence of choices, $0^{f(n)}$. Thus, part (b) of the theorem has been proved.

Let us now prove (c), which is methodologically the more interesting result. The proof of (c), although quite simple and clear, involves a powerful general method for simulating space-bounded machines, which can be called *the reachability method.* This method will be used next to prove two far more interesting results.

We are given a k-string nondeterministic machine M with input and output, which decides L within space $f(n)$. We must find a deterministic method for simulating the nondeterministic computation of M on input x within time $c^{\log n + f(n)}$, where $n = |x|$, for some constant c depending on M. At this point it is useful to recall the concept of a *configuration* of M: A configuration is intuitively a "snapshot" of the computation of M on the given input x. For a k-string machine, it is a $2k+1$-tuple $(q, w_1, u_1, \ldots, w_k, u_k)$ recording the state, the strings, and the head positions. Now, for a machine with input and output such as M, the second and third component of the configuration will always

spell $\triangleright x$. Also, for a machine deciding a language, such as M, the write-only output string is not interesting. And all $k-2$ other strings will be of length at most $f(n)$. Thus, a configuration can be represented by a $2k-2$-tuple $(q, i, w_2, u_2, \ldots, w_{k-1}, u_{k-1})$, where i is an integer $0 \leq i \leq n = |x|$ recording the position of the first cursor on the input string (always spelling $\triangleright x$).

How many configurations are there? There are $|K|$ choices for the first component (the state), $n+1$ choices for i, and fewer than $|\Sigma|^{(2k-2)f(n)}$ choices for all the remaining strings together. All told, the total number of configurations of M when operating on an input of length n is at most $nc_1^{f(n)} = c_1^{\log n + f(n)}$ for some constant c_1 depending only on M.

Define next the *configuration graph* of M on input x, denoted $G(M, x)$, to be the graph that has as its set of nodes the set of all possible configurations, and an edge between two configurations C_1 and C_2 if and only if $C_1 \xrightarrow{M} C_2$. It is immediate now that *telling whether $x \in L$ is equivalent to deciding whether there is a path in the configuration graph $G(M, x)$ from the initial configuration $C_0 = (s, 0, \triangleright, \epsilon, \ldots, \epsilon)$ to some configuration of the form $C = (\text{"yes"}, i, \ldots)$.*

In other words, we have reduced the problem of deciding whether $x \in L$ to the REACHABILITY problem of a graph with $c_1^{\log n + f(n)}$ nodes. Since we know that there is a polynomial-time algorithm for graph reachability (this is the first fact we established in this book, back in Section 1.1), the simulation takes $c_2 c_1^{2(\log n + f(n))}$ steps, where the polynomial bound for the connectivity problem is taken to be a generous $(c_2 n^2)$. By taking $c = c_2 c_1^2$, we have achieved the required time bound.

But we have not specified exactly how the polynomial algorithm for reachability is used in this context. There are several options. One is to construct explicitly the adjacency matrix of $G(M, x)$ on a string of the simulating machine, and then run the reachability algorithm. A more elegant idea is to run the reachability algorithm *without first constructing the adjacency list.* Instead, whenever we need to tell whether (C, C') is an edge of $G(M, x)$ for two configurations C and C', we invoke a simple routine that decides whether two configurations yield one another. In other words, in this latter approach the configuration graph is *given implicitly* by x. Given two such configurations, say C and C', we just need the symbols scanned in each string of C to determine whether indeed C' can be yielded from C. This is trivial to do for all strings except for the input string, where we only know the cursor position i. To recover the ith symbol of x, we simply look it up from the input x of the machine, counting symbols from the left of the input string, increasing a binary counter (on a separate string) until it becomes i. □

Combining the information from Theorem 7.4, we obtain the following tower of class inclusions:

Corollary: L ⊆ NL ⊆ P ⊆ NP ⊆ PSPACE. □

Now, we know from the space hierarchy theorem (Theorem 7.2) that **L** is a *proper subset* of **PSPACE**. Thus, at least one of the four inclusions of the Corollary must be proper. This brings about yet another amusing way of expressing the frustrations of complexity theory: Although we strongly suspect that the *conjunction* of these four conjectured proper inclusions is true, we are currently sure only of their *disjunction!*

Nondeterministic Space

The reachability method for simulating nondeterministic space has two other applications. The first concerns a rather surprising result on the simulation of nondeterministic space by deterministic space. It is immediate from (c) of Theorem 7.4 that $\mathbf{NSPACE}(f(n)) \subseteq \mathbf{SPACE}(k^{\log n + f(n)})$. Is there a better than exponential way of simulating nondeterministic space by deterministic space, or is it the case that nondeterminism in space is exponentially more powerful than determinism (as we suspect in the case of time)? We shall next prove that the simulating deterministic space *need only be quadratic.*

We shall establish this by the reachability method. To apply the method, we prove first an algorithmic result about solving graph reachability in limited deterministic space. The breadth-first search and depth-first search techniques discussed in Section 1.1 both require in the worst case space at least n, the number of nodes in the graph. As we show next, there is an ingenious "middle-first search" technique which, while wasteful in time, succeeds in limiting the space needed.

Theorem 7.5 (Savitch's Theorem): REACHABILITY$\in \mathbf{SPACE}(\log^2 n)$.

Proof: Let G be a graph with n nodes, let x and y be nodes of G, and let $i \geq 0$. We say that predicate PATH(x, y, i) holds if the following is true: There is a path from x to y in G, of length at most 2^i. Notice immediately that, since a path in G need be at most n long, we can solve reachability in G if we can compute whether PATH$(x, y, \lceil \log n \rceil)$ for any two given nodes of G.

We shall design a Turing machine, with two working strings besides its input string, which decides whether PATH(x, y, i). The adjacency matrix of G is given at the input string. We assume that the first work string contains nodes x and y and integer i, all in binary. The first work string will typically contain several triples, of which the (x, y, i) one will be the leftmost. The other work string will be used as scratch space –$\mathcal{O}(\log n)$ space will suffice there.

We now describe how our machine decides whether PATH(x, y, i). If $i = 0$, we can tell whether x and y are connected via a path of length at most $2^i = 1$ by simply checking whether $x = y$, or whether x and y are adjacent by looking at the input. This takes care of the $i = 0$ case. If $i \geq 1$, then we compute PATH(x, y, i) by the following recursive algorithm:

for all nodes z test whether PATH$(x, z, i-1)$ and PATH$(z, y, i-1)$

This program implements a very simple idea: *Any path of length 2^i from x to y has a midpoint z*[†]*, and both x and y are at most 2^{i-1} away from this midpoint.* To implement this elegant idea in a space-efficient manner, we generate all nodes z, one after the other, reusing space. Once a new z is generated, we add a triple $(x, z, i-1)$ to the main work string and start working on this problem, recursively. If a negative answer to PATH$(x, z, i-1)$ is obtained, we erase this triple and move to the next z. If a positive answer is returned, we erase the triple $(x, z, i-1)$, write the triple $(z, y, i-1)$ on the work string (we consult (x, y, i), the next triple to the left, to obtain y), and work on deciding whether PATH$(z, y, i-1)$. If this is negative, we erase the triple and try the next z; if it is positive, we detect by comparing with triple (x, y, i) to the left that this is the second recursive call, and return a positive answer to PATH(x, y, i). Notice that the first working string of the machine acts like a *stack of activation records* to implement the recursion indicated above.

It is clear that this algorithm implements the recursive one displayed above, and thus correctly solves PATH(x, y, i). The first work string contains at any moment $\lceil \log n \rceil$ or fewer triples, each of length at most $3 \log n$. And in order to solve REACHABILITY, we start this algorithm with $(x, y, \lceil \log n \rceil)$ written on the main work string. The proof is complete. □

Savitch's theorem yields, via the reachability method, an important complexity result:

Corollary: **NSPACE**$(f(n)) \subseteq$ **SPACE**$(f^2(n))$ for any proper complexity function $f(n) \geq \log n$.

Proof: To simulate an $f(n)$-space bounded nondeterministic machine M on input x, $|x| = n$, we simply run the algorithm in the proof of Theorem 7.5 on the configuration graph of M on input x. Notice that, as usual, the only interaction of this algorithm with the input was in checking whether two nodes are connected (for the $i = 0$ base case). In our new algorithm, each time such a check arises we decide instead, by examining the input and based on the transition function of the simulated machine, whether two nodes of the configuration graph are connected by an edge. Since the configuration graph has $c^{f(n)}$ nodes for some constant c, $\mathcal{O}(f^2(n))$ space suffices. □

This result immediately implies that **PSPACE** = **NPSPACE**, a result which strongly suggests that nondeterminism is less powerful with respect to space than it is with respect to time. Our next result is further evidence of this: We show that *nondeterministic space classes are closed under complement* (whereas it is very doubtful that nondeterministic *time* classes are closed under complement). Once again, we first prove an algorithmic result for a variant of the reachability problem.

[†] In honor of Zeno of Elea, see the references.

Recall that we can solve REACHABILITY nondeterministically in space $\log n$. We next show that we can solve in nondeterministic space $\log n$ an important extension of REACHABILITY: Computing the number of nodes reachable from x. Notice that this is equivalent to counting the nodes *not* reachable from x (just subtract from n). That is, the counting problem and its "complement" are identical. So, it makes sense that this problem is instrumental for showing that nondeterminism in space is closed under complement (see the corollary below).

We must first define what it means for a nondeterministic Turing machine to *compute* a function F from strings to strings. It simply means that, on input x, each computation of the machine either outputs the correct answer $F(x)$, or ends up in state "no". Of course, we insist that at least one computation ends up with $F(x)$—otherwise all functions would be trivial to compute... In other words, we require that all "successful" computations agree on their output, whereas all "unsuccessful" ones realize that they are unsuccessful. Such machine observes a space bound $f(n)$, as always, if at halting all strings (except for the input and output ones) are of length at most $f(|x|)$—and this must be true of all computations, successful or not.

Theorem 7.6 (The Immerman-Szelepscényi Theorem): Given a graph G and a node x, the number of nodes reachable from x in G can be computed by a nondeterministic Turing machine within space $\log n$.

Proof: As we did with Theorem 7.5, we shall first describe the basic algorithmic idea behind the machine that achieves this. The algorithm has four nested loops. Although each of the loops is fairly simple, nondeterminism makes their interaction quite subtle.

The outermost loop computes iteratively $|S(1)|, |S(2), | \ldots |S(n-1)|$, where $S(k)$ is the set of nodes in G that can be reached from x by paths of length k or less. Obviously, $|S(n-1)|$ is the desired answer, where n is the number of nodes of G. Once $|S(k-1)|$ is produced, we start the computation of $|S(k)|$. Initially, we know $|S(0)| = 1$. Thus, the outermost loop is this:

$|S(0)| := 1$; for $k = 1, 2, \ldots n-1$ do: compute $|S(k)|$ from $|S(k-1)|$

The computation of $|S(k)|$ uses $|S(k-1)|$ (but, fortunately, no previous $|S(j)|$'s). A count ℓ is initialized to zero. We then examine all nodes of G, one by one, in numerical order, reusing space. If a node is found to be in $S(k)$, ℓ is incremented. At the end, ℓ contains the true value of $|S(k)|$:

$\ell := 0$; for each node $u = 1, 2, \ldots, n$ do: if $u \in S(k)$ then $\ell := \ell + 1$

But of course, we have not said how to decide whether $u \in S(k)$. This is done by the third loop. In this loop, we go over all nodes v of G, again in numerical order and reusing space. If a node v is in $S(k-1)$, we increment a counter m of the members of $S(k-1)$ encountered so far. Then it is checked

whether $u = v$ or there is an edge from v to u (we use $G(v, u)$ in the description below to mean that either $u = v$ or there is an arc from v to u in G). If so, we have established that $u \in S(k)$, and we set the variable "reply" to true. If we reach the end without showing $u \in S(k)$, we report that $u \notin S(k)$. *If, however, we discover, by comparing m with the known value of $|S(k-1)|$, that we have not accounted for all members of $S(k-1)$* (presumably due to our imperfect, nondeterministic way of telling whether $v \in S(k-1)$) *then we give up* by entering state "no" (recall what it means for a nondeterministic Turing machine to compute a function). The present computation will not affect the end result.

$m := 0$; reply:=false; for each node $v = 1, 2, \ldots, n$ repeat:
if $v \in S(k-1)$ then $m := m + 1$; if furthermore $G(v, u)$ then reply:=true;
if in the end $m < |S(k-1)|$ then "no" (give up), else return reply

But how do we tell whether $v \in S(k-1)$? The answer is yet another loop, but this time one familiar from the nondeterministic $\log n$-space algorithm for REACHABILITY in Example 2.10: By using nondeterminism, we start at node x, guess $k-1$ nodes, and for each we check that either it is the same with the previous one, or there is an arc from the previous one to it. We report that there is a path from x to v if the last node is v:

$w_0 := x$; for $p = 1, \ldots, k-1$ do:
guess a node w_p and check that $G(w_{p-1}, w_p)$ (if not, give up);
if $w_{k-1} = v$ then report $v \in S(k-1)$, otherwise give up

This completes the description of the algorithm. It is easy to see that this algorithm can be implemented in a $\log n$ space-bounded Turing machine M. M has a separate string holding each of the nine variables k, $|S(k-1)|$, ℓ, u, m, v, p, w_p, and w_{p-1}, plus an input and an output string. These integers need only be incremented by one, and compared to each other and with the nodes in the input. And they are all bounded by n.

We shall next prove that this algorithm is correct, that is, it correctly computes $|S(k)|$, for all k. The claim is perfectly true when $k = 0$. For general $k \geq 1$, consider the value of $|S(k)|$ computed by a successful computation (that is, one that never rejected having discovered that $m < |S(k-1)|$); all we have to prove is that counter ℓ is incremented if and only if the current u is in $S(k)$. Since the loop on m and v has not rejected, m equals $|S(k-1)|$ (by induction, the correct value thereof). This means that all $v \in S(k-1)$ were verified as such (since the innermost loop never makes false positive errors, that is, it never decides that a path from x to v exists where it does not), and thus the variable "reply" accurately records whether $u \in S(k)$, and it is this variable that determines whether ℓ is incremented. Finally, it is easy to see that at least one successful computation exists (the one that correctly guesses the members

of $S(k-1)$ and the paths to each). The proof is complete. □

There is a complexity consequence of this result which is as immediate as it is important:

Corollary: If $f \geq \log n$ is a proper complexity function, then **NSPACE**$(f(n))$ = **coNSPACE**$(f(n))$.

Proof: Suppose that $L \in$ **NSPACE**$(f(n))$, decided by an $f(n)$ space bounded nondeterministic Turing machine M; we shall show that there is an $f(n)$ space-bounded nondeterministic Turing machine $\overline{M}$ deciding $\overline{L}$. On input x, $\overline{M}$ runs the algorithm in the proof of Theorem 7.5 on the configuration graph of M on input x. As usual, each time the algorithm needs to tell whether two configurations are connected, $\overline{M}$ decides this on the basis of x and the transition function of M. Finally, if, while running this algorithm, $\overline{M}$ discovers that an accepting configuration u is found to be in $S(k)$, for any value of k, then it halts and rejects; otherwise, if $|S(n-1)|$ is computed and no accepting configuration has been encountered, $\overline{M}$ accepts. □

7.4 NOTES, REFERENCES, AND PROBLEMS

7.4.1 The quotation in the header of Part III is from

- J. Edmonds "Systems of distinct representatives and linear algebra," and "Optimum branchings," *J. Res. National Bureau of Standards, Part B, 17B*, 4, pp. 241–245 and 233–240, 1966–1967.

7.4.2 Although complexity-like subclasses of recursive functions were studied in the 1950's

- A. Grzegorczyk "Some classes of recursive functions," *Rosprawy Matematyzne 4*, Math. Inst. of the Polish Academy of Sciences, 1953;
- M. O. Rabin "Degree of difficulty of computing a function and a partial ordering of recursive sets," Tech. Rep. No 2, Hebrew Univ., 1960,

and several authors in the 1950's and 60's had discussed informally or elliptically complexity issues (sometimes going as far as formulating the $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ problem, see the references by Edmonds above), the systematic and formal study of time and space complexity classes started with these pioneering papers:

- J. Hartmanis and R. E. Stearns "On the computational complexity of algorithms," *Transactions of the AMS, 117*, pp. 285–306, 1965, and
- J. Hartmanis, P. L. Lewis II, and R. E. Stearns "Hierarchies of memory-limited computations," *Proc. 6th Annual IEEE Symp. on Switching Circuit Theory and Logic Design*, pp. 179–190, 1965.

The foundations of complexity theory were laid in this work, and Theorems 7.1 and 7.2, as well as Theorems 2.2 and 2.3, were proved there.

7.4.3 Problem: (a) Show that if $f(n)$ and $g(n)$ are proper complexity functions, then so are $f(g)$, $f + g$, $f \cdot g$, and 2^g.

(b) Show that the following are proper complexity functions: (i) $\log n^2$, (ii) $n \log n$, (iii) n^2, (iv) $n^3 + 3n$, (v) 2^n, (vi) 2^{n^2}, (vii) $\sqrt{n}$, and (viii) $n!$.

Our notion of proper complexity functions is perhaps the simplest of the many possible formulations of what has been called in the literature *honesty*, and *space- or time-constructibility*, each with several subtle variants.

7.4.4 Problem: Let C be a *class of functions* from nonnegative integers to nonnegative integers. We say that C is closed under *left polynomial composition* if $f(n) \in C$ implies $p(f(n)) = \mathcal{O}(g(n))$ for some $g(n) \in C$, for all polynomials $p(n)$. We say that C is closed under *right polynomial composition* if $f(n) \in C$ implies $f(p(n)) = \mathcal{O}(g(n))$ for some $g(n) \in C$, for all polynomials $p(n)$.

Intuitively, the first closure property implies that the corresponding complexity class is "computational model-independent, " that is, it is robust under reasonable changes in the underlying model of computation (from RAM's to Turing machines, to multistring Turing machines, etc.) while closure under right polynomial composition suggests closure under *reductions* (see the next chapter).

Which of the following classes of functions are closed under left polynomial composition, and which under right polynomial composition?
(a) $\{n^k : k > 0\}$.
(b) $\{k \cdot n : k > 0\}$.
(c) $\{k^n : k > 0\}$.
(d) $\{2^{n^k} : k > 0\}$.
(e) $\{\log^k n : k > 0\}$.
(f) $\{\log n\}$.

7.4.5 Problem: Show that **P** is closed under union and intersection. Repeat for **NP**.

7.4.6 Problem: Define the *Kleene star* of a language L to be $L^* = \{x_1 \dots x_k : k \geq 0; x_1, \dots, x_k \in L\}$ (notice that our notation Σ^* is compatible with this definition). Show that **NP** is closed under Kleene star. Repeat for **P**. (This last one is a little less obvious.)

7.4.7 Problem: Show that $\mathbf{NP} \neq \mathbf{SPACE}(n)$. (We have no idea if one includes the other, but we know they are different! Obviously, closure under some operation must be used.)

7.4.8 Problem: Refine the time hierarchy Theorem 7.1 to show that if $f(n)$ is a proper function then $\mathbf{TIME}(f(n))$ is a proper subset of $\mathbf{TIME}(f(n)\log^2 f(n))$. (Use a two-string universal Turing machine U_f, recall Problem 2.8.9. In fact, any function growing faster than $\log f(n)$ would do as a multiplier.)

7.4.9 Problem: Prove the space hierarchy Theorem 7.2. Is the $\log f(n)$ factor needed?

7.4.10 We also have *nondeterministic* time and space hierarchy theorems; see

- S. A. Cook "A hierarchy for nondeterministic time complexity," J.CSS, 7, 4, pp. 343–353, 1973, and
- J. I. Seiferas, M. J. Fischer, and A. R. Meyer "Refinements of nondeterministic time and space hierarchies," *Proc. 14th IEEE Symp. on the Foundations of Computer Science*, pp. 130–137, 1973.

and Problem 20.2.5. This hierarchy is in fact *tighter* than its deterministic counterpart; the reason can be traced to Problem 2.8.17: A nondeterministic Turing machine can be assumed to have only two strings with no harm to its time performance.

7.4.11 Problem: State and prove the gap theorem for space. Also, prove that the gap theorem holds when we substitute any recursive function for 2^n.

The gap theorem is from

- B. A. Trakhtenbrot "Turing computations with logarithmic delay," *Algebra i Logika, 3*, 4, pp. 33–48, 1964,

and was independently rediscovered in

- A. Borodin "Computational complexity and the existence of complexity gaps," *J.ACM 19*, 1, pp. 158–174, 1972.

7.4.12 Blum complexity. Time and space are only two examples of "complexity measures" for computations. In general, suppose that we have a function Φ, possibly undefined on many arguments, mapping Turing machine-input pairs to the nonnegative integers. Suppose Φ is such that the following two axioms hold:

Axiom 1: $\Phi(M, x)$ is defined if and only if $M(x)$ is defined.

Axiom 2: It is decidable, given M, x, and k, whether $\Phi(M, x) = k$.
Then Φ is called a *complexity measure*. This elegant formulation of complexity was developed in

- M. Blum "A machine-independent theory of the complexity of recursive functions," *J.ACM 14*, 2, pp. 322–336, 1967.

(a) Show that space and time are complexity measures. (Notice that in this context we do not maximize space and time over all strings of the same length, but leave the dependence on individual strings.) Repeat for nondeterministic space and time.

(b) Show that *ink* (the number of times during a computation that a symbol has to be overwritten by a different symbol) is a complexity measure.

(c) Show that *reversals* (the number of times during a computation that the cursor must change direction of motion) is a complexity measure.

(d) Show that *carbon* (the number of times during a computation that a symbol has to be overwritten with *the same* symbol) is *not* a complexity measure.

We have seen in this chapter theorems that relate the four "ordinary" complexity measures in (a) above. These results are part of a bigger pattern:

(e) Show that, if Φ and Φ' are complexity measures, then there is a recursive function b such that for all M and all x $\Phi(M, x) \leq b(\Phi(M, x))$.

(f) Prove that the gap theorem holds for any complexity measure.

7.4.13 Speedup theorem. In principle, the theory of algorithms and complexity strives to determine for each computational problem the fastest algorithm that solves it. In our Turing machine model, with its arbitrarily large alphabet and state space, we know that there is no single fastest algorithm, as any Turing machine can be sped up by a constant amount (recall Theorem 2.2). The following result, from

- M. Blum "On effective procedures for speeding up algorithms," *J.ACM 18*, 2, pp. 290–305, 1971,

shows that, for some languages, arbitrarily large speedups are possible:

Prove that there is a language L such that, for every Turing machine that decides L in space $f(n)$ there is a Turing machine deciding L in space $g(n)$, where $g(n) \leq \log f(n)$ for almost all n. (Define the function T by $T(1) = 2$, and $T(n+1) = 2^{T(n)}$. That is, $T(n)$ is a *tower of n twos*. Construct L carefully so that (a) for all k there are Turing machines that decide it and use less than $T(n-k)$ space on inputs of length n, whereas if $L(M_i) = L$ then M_i requires at least $T(n-i)$ space for some input of length n.)

Needless to say, the result holds for any slow recursive function replacing $\log n$, as well as for any complexity measure.

7.4.14 For more on Blum complexity see

- J. Seiferas, "Machine-independent complexity theory," pp. 163–186 in *The Handbook of Theoretical Computer Science, vol. I: Algorithms and Complexity*, edited by J. van Leeuwen, MIT Press, Cambridge, Massachusetts, 1990.

7.4.15 In order to reach point 0 on the real line from point 1, we must first arrive at the *midpoint* $\frac{1}{2}$. In order to traverse the remaining distance, we must pass the midpoint $\frac{1}{4}$. And so on. The ancient Greek philosopher Zeno of Elea considered this infinite sequence of midpoints as evidence that *motion is impossible* (apparently he was missing the subtle point, that an infinity of positive real numbers may have a finite sum). Savitch's algorithm is based on the fact that, in a discrete context, the sequence of midpoints is only logarithmically long.

7.4.16 Theorem 7.5 is due to Walt Savitch:

- W. J. Savitch "Relationship between nondeterministic and deterministic tape classes," *J.CSS, 4*, pp. 177-192, 1970.

The Immerman-Szelepcsényi theorem (Theorem 7.6) was shown independently by

- N. Immerman "Nondeterministic space is closed under complementation," *SIAM J. Computing, 17*, pp. 935–938, 1988, and
- R. Szelepcsényi "The method of forcing for nondeterministic automata," *Bull. of the EATCS, 33*, pp. 96–100, 1987.

7.4.17 Time vs. space. We know that $\mathbf{TIME}(f(n)) \subseteq \mathbf{SPACE}(f(n))$; it is shown in

- J. E. Hopcroft, W. J. Paul, and L. G. Valiant "On time vs. space and related problems," *Proc. 16th Annual IEEE Symp. on the Foundations of Computer Science*, pp. 57–64, 1975

that, for proper f, $\mathbf{TIME}(f(n)) \subseteq \mathbf{SPACE}(\frac{f(n)}{\log f(n)})$. The proof standardizes the computation of a $f(n)$ time-bounded Turing machine on input x so that the strings are divided in *blocks* of size $\sqrt{f(|x|)}$. The machine is *block-respecting*, that is, block boundaries are crossed only in steps that are integer multiples of $\sqrt{f(|x|)}$.

(a) Show that any $f(n)$ time-bounded k-string Turing machine can be made block-respecting with no increase in time complexity.

The operation of a block-respecting k-string Turing machine M on input x can be captured by a graph $G_M(x)$ with $\sqrt{f(|x|)}$ nodes each of indegree and outdegree $\mathcal{O}(k)$. The nodes stand for computation segments of length $\sqrt{f(|x|)}$, while edges signify "information flow." That is, (B, B') is an edge if and only if information from segment B is necessary for carrying out the computation in segment B'.

(b) Define precisely the graph $G_M(x)$. Argue that it is acyclic.

Carrying out the computation of M on x in limited space can now be thought of as "computing" the graph $G_M(x)$, using "registers" of size $\sqrt{f(|x|)}$. The point is to do it with as few registers as possible, perhaps using recomputation.

(c) Define precisely what is a computation of a directed acyclic graph by R registers. Show that any directed acyclic graph with n nodes and bounded indegree and outdegree can be computed with $\mathcal{O}(\frac{n}{\log n})$ registers. (This is by far the hardest part of the proof, and it involves a sophisticated "divide-and-conquer" technique.)

(d) Conclude that $\mathbf{TIME}(f(n)) \subseteq \mathbf{SPACE}(\frac{f(n)}{\log f(n)})$. (Use (c) above to guess a computation on $G_M(x)$, and apply Savitch's theorem).

(e) Show that for one-string machines M, $G_M(x)$ is always planar. What can you conclude about time and space in one-string machines? (Prove first that planar graphs can be computed with $\mathcal{O}(\sqrt{n})$ registers.)

CHAPTER 8

REDUCTIONS AND COMPLETENESS

Certain problems capture the difficulty of a whole complexity class. Logic plays a central role in this fascinating phenomenon.

8.1 REDUCTIONS

Like all complexity classes, **NP** contains an infinity of languages. Of the problems and languages we have seen so far in this book, **NP** contains TSP (D) (recall Section 1.3) and the SAT problem for Boolean expressions (recall Section 4.3). In addition, **NP** certainly contains REACHABILITY, defined in Section 1.1, and CIRCUIT VALUE from Section 4.3 (both are in **P**, and thus certainly in **NP**). It is intuitively clear, however, that the former two problems are somehow more worthy representatives of **NP** than the latter two. They seem to capture more faithfully the power and complexity of **NP**, they are not known (or believed) to be in **P** like the other two. We shall now introduce concepts that make this intuition precise and mathematically provable.

What we need is a precise notion of what it means for a problem to be *at least as hard as* another. We propose *reduction* (recall the discussion in Sections 1.2 and 3.2) as this concept. That is, we shall be prepared to say that problem A is at least as hard as problem B if B reduces to A. Recall what "reduces" means. We say that B reduces to A if there is a transformation R which, for every input x of B, produces an equivalent input $R(x)$ of A. Here by "equivalent" we mean that the answer to $R(x)$ considered as an input for A, "yes" or "no," is a correct answer to x, considered as an input of B. In other words, to solve B on input x we just have to compute $R(x)$ and solve A on it (see Figure 8.1).

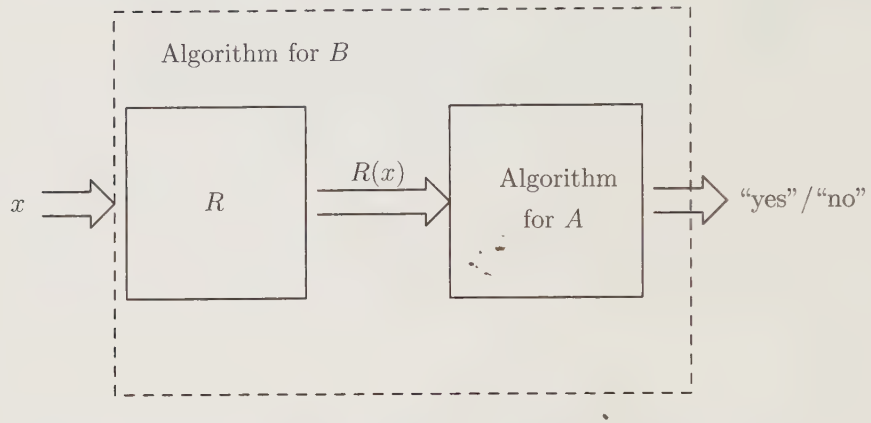


Figure 8-1. Reduction from B to A.

If the scenario in Figure 8.1 is possible, it seems reasonable to say that A is at least as hard as B. With one proviso: *That R should not be fantastically hard to compute.* If we do not limit the complexity of computing R, we could arrive at absurdities such as TSP (D) reduced to REACHABILITY, and thus REACHABILITY being harder than TSP (D)! Indeed, given any instance x of TSP (D) (that is, a distance matrix and a budget), we can apply the following reduction: Examine all tours; if one of them is cheaper than the budget, then $R(x)$ is the two-node graph consisting of a single edge from 1 to 2. Otherwise, it is the two-node graph with no edges. Notice that, indeed, $R(x)$ is a "yes" instance of REACHABILITY if and only if x was a "yes" instance of TSP (D). The flaw is, of course, that R is an exponential-time algorithm.

Definition 8.1: As we pointed out above, for our concept of reduction to be meaningful, it should involve the weakest computation possible. *We shall adopt $\log n$ space-bounded reduction as our notion of "efficient reduction."* That is, we say that language L_1 *is reducible to* L_2 if there is a function R from strings to strings computable by a deterministic Turing machine in space $\mathcal{O}(\log n)$ such that for all inputs x the following is true: $x \in L_1$ if and only if $R(x) \in L_2$. R is called a *reduction* from L_1 to L_2. □

Since our focal problems in complexity involve the comparisons of time classes, it is important to note that reductions are *polynomial-time algorithms.*

Proposition 8.1: If R is a reduction computed by Turing machine M, then for all inputs x M halts after a polynomial number of steps.

Proof: There are $\mathcal{O}(nc^{\log n})$ possible configurations for M on input x, where $n = |x|$. Since the machine is deterministic, no configuration can be repeated in the computation (because such repetition means the machine does not halt).

Thus, the computation is of length at most $\mathcal{O}(n^k)$ for some k. □

Needless to say, since the output string $R(x)$ is computed in polynomial time, its length is also polynomial (since at most one new symbol can be output at each step). We next see several interesting examples of reductions.

Example 8.1: Recall the problem HAMILTON PATH briefly discussed in Example 5.12. It asks, given a graph, whether there is a path that visits each node exactly once. Although HAMILTON PATH is a very hard problem, we next show that SAT (the problem of telling whether a given Boolean expression has a satisfying truth assignment) is at least as hard: We show that HAMILTON PATH can be reduced to SAT. We describe the reduction next.

Suppose that we are given a graph G. We shall construct a Boolean expression $R(G)$ such that $R(G)$ is satisfiable if and only if G has a Hamilton path. Suppose that G has n nodes, $1, 2, \ldots, n$. Then $R(G)$ will have n^2 Boolean variables, $x_{ij} : 1 \leq i, j, \leq n$. Informally, variable x_{ij} will represent the fact "node j is the ith node in the Hamilton path," which of course may be either true or false. $R(G)$ will be in conjunctive normal form, so we shall describe its clauses. The clauses will spell out all requirements on the x_{ij}'s that are sufficient to guarantee that they encode a true Hamilton path. To start, node j must appear in the path; this is captured by the clause $(x_{1j} \vee x_{2j} \vee \ldots \vee x_{nj})$; we have such a clause for each j. But node j cannot appear both ith and kth: This is expressed by clause $(\neg x_{ij} \vee \neg x_{kj})$, repeated for all values of j, and $i \neq k$. Conversely, some node must be ith, thus we add the clause $(x_{i1} \vee x_{i2} \vee \ldots \vee x_{in})$ for each i; and no two nodes should be ith, or $(\neg x_{ij} \vee \neg x_{ik})$ for all i, and all $j \neq k$. Finally, for each pair (i, j) which is *not* an edge of G, it must not be the case that j comes right after i in the Hamilton path; therefore the following clauses are added for each pair (i, j) not in G and for $k = 1, \ldots, n-1$: $(\neg x_{ki} \vee \neg x_{k+1,j})$. This completes the construction. Expression $R(G)$ is the conjunction of all these clauses.

We claim that R is a reduction from HAMILTON PATH to SAT. To prove our claim, we have to establish two things: That for any graph G, expression $R(G)$ has a satisfying truth assignment if and only if G has a Hamilton path; and that R can be computed in space $\log n$.

Suppose that $R(G)$ has a satisfying truth assignment T. Since T satisfies all clauses of $R(G)$, it must be the case that, for each j there exists a unique i such that $T(x_{ij}) = \textbf{true}$, otherwise the clauses of the form $(x_{1j} \vee x_{2j} \vee \ldots \vee x_{nj})$ and $(\neg x_{ij} \vee \neg x_{kj})$ cannot all be satisfied. Similarly, clauses $(x_{i1} \vee x_{i2} \vee \ldots \vee x_{in})$ and $(\neg x_{ij} \vee \neg x_{ik})$ guarantee that for each i there exists a unique j such that $T(x_{ij}) = \textbf{true}$. Hence, T really represents a *permutation* $\pi(1), \ldots, \pi(n)$ of the nodes of G, where $\pi(i) = j$ if and only if $T(x_{ij}) = \textbf{true}$. However, clauses $(\neg x_{k,i} \vee \neg x_{k+1,j})$ where (i, j) is not an edge of G and $k = 1, \ldots, n-1$ guarantee that, for all k, $(\pi(k), \pi(k+1))$ is an edge of G. This means that $(\pi(1), \pi(2), \ldots, \pi(n))$ is a

Hamilton path of G.

Conversely, suppose that G has a Hamilton path $(\pi(1), \pi(2), \ldots, \pi(n))$, where π is a permutation. Then it is clear that the truth assignment $T(x_{ij}) =$ **true** if $\pi(i) = j$, and $T(x_{ij}) =$ **false** if $\pi(i) \neq j$, satisfies all clauses of $R(G)$.

We still have to show that R can be computed in space $\log n$. Given G as an input, a Turing machine M outputs $R(G)$ as follows: First it writes n, the number of nodes of G, in binary, and, based on n it generates in its output tape, one by one, the clauses that do not depend on the graph (the first four groups in the description of $R(G)$). To this end, M just needs three counters, i, j, and k, to help construct the indices of the variables in the clauses. For the last group, the one that depends on G, M again generates one by one in its work string all clauses of the form $(\neg x_{ki} \vee \neg x_{k+1,j})$ for $k = 1, \ldots, n-1$; after such a clause is generated, M looks at its input to see whether (i, j) is an edge of G, and if it is not, then it outputs the clause. This completes our proof that HAMILTON PATH can be reduced to SAT.

We shall see many more reductions in this book; however, the present reduction is one of the simplest and clearest that we shall encounter. The clauses produced express in a straightforward and natural way the requirements of HAMILTON PATH, and the proof need only check that this translation is indeed accurate. Since SAT, the "target" problem, is a problem inspired from logic, one should not be surprised that it can "express" other problems quite readily: After all, expressiveness is logic's strongest suit. □

Example 8.2: We can also reduce REACHABILITY to CIRCUIT VALUE (another problem inspired by logic). We are given a graph G and wish to construct a variable-free circuit $R(G)$ such that the output of $R(G)$ is **true** if and only if there is a path from node 1 to node n in G.

The gates of $R(G)$ are of the form g_{ijk} with $1 \leq i, j \leq n$ and $0 \leq k \leq n$, and h_{ijk} with $1 \leq i, j, k \leq n$. Intuitively, g_{ijk} is **true** if and only if there is a path in G from node i to node j *not using any intermediate node bigger than* k. On the other hand, h_{ijk} will be **true** if and only if there is a path in G from node i to node j again not using intermediate nodes bigger than k, *but using* k *as an intermediate node*. We shall next describe each gate's sort and predecessors. For $k = 0$, all g_{ij0} gates are input gates (recall that there are no h_{ij0} gates). In particular, g_{ij0} is a **true** gate if and only if either $i = j$ or (i, j) is an edge of G, and it is a **false** gate otherwise. This is how the structure of G is reflected in $R(G)$. For $k = 1, \ldots, n$, h_{ijk} is an AND gate (that is, $s(h_{ijk}) = \wedge$), and its predecessors are $g_{i,k,k-1}$ and $g_{k,j,k-1}$ (that is, there are edges $(g_{i,k,k-1}, h_{ijk})$ and $(g_{k,j,k-1}, h_{ijk})$ in $R(G)$). Also, for $k = 1, \ldots, n$, g_{ijk} is an OR gate, and there are edges $(g_{i,j,k-1}, g_{ijk})$ and (h_{ijk}, g_{ijk}) in $R(G)$. Finally, g_{1nn} is the output gate. This completes the description of circuit $R(G)$.

It is easy to see that $R(G)$ is indeed a legitimate variable-free circuit, whose

gates can be renamed $1, 2, \ldots, 2n^3 + n^2$ (in nondecreasing order of the third index, say) so that edges go from lower-numbered gates to higher-numbered ones, and indegrees are in accordance to sorts. (Notice that there are no NOT gates in $R(G)$.) We shall next show that the value of the output gate of $R(G)$ is **true** if and only if there is a path from 1 to n in G.

We shall prove by induction on k that the values of the gates are indeed the informal meanings described above. The claim is true when $k = 0$, and if it is true up to $k - 1$, the definitions of h_{ijk} as $(g_{i,k,k-1} \wedge g_{k,j,k-1})$ and of h_{ijk} as $(h_{ijk} \vee g_{i,j,k-1})$ guarantee it to be true for k as well. Hence, the output gate g_{1nn} is **true** if and only if there is a path from 1 to n using no intermediate nodes numbered above n (of which there is none), that is, if and only if there is a path from 1 to n in G.

Furthermore, R can be computed in $\log n$ space. The machine would again go over all possible indices i, j, and k, and output the appropriate edges and sorts for the variables. Hence R is a reduction from REACHABILITY to CIRCUIT VALUE.

It is instructive to notice that circuit $R(G)$ is derived from a polynomial-time algorithm for REACHABILITY, namely the well-known Floyd-Warshall algorithm. As we shall see soon, rendering polynomial algorithms as variable-free circuits is a quite general pattern. It is remarkable that the circuit uses no NOT gates, and is thus a *monotone circuit* (see Problems 4.4.13 and 8.4.7). Finally, notice that the circuit constructed has *depth* (length of the longest path from an input to an output gate) that is linear in n. In Chapter 15 we shall exhibit a much "shallower" circuit for the same problem. □

Example 8.3: We can also reduce CIRCUIT SAT (recall Section 4.3) to SAT. We are given a circuit C, and wish to produce a Boolean expression $R(C)$ such that $R(C)$ is satisfiable if and only if C is satisfiable. But this is not hard to do, since expressions and circuits are different ways of representing Boolean functions, and translations back and forth are easy. The variables of $R(C)$ will contain all variables appearing in C, and in addition, for each gate g of C we are going to have a variable in $R(C)$, also denoted g. For each gate of C we shall generate certain clauses of $R(C)$. If g is a variable gate, say corresponding to variable x, then we add the two clauses $(\neg g \vee x)$ and $(g \vee \neg x)$. Notice that any truth assignment T that satisfies both clauses must have $T(g) = T(x)$; to put it otherwise, $(\neg g \vee x) \wedge (g \vee \neg x)$ is the conjunctive normal form of $g \Leftrightarrow x$. If g is a **true** gate, then we add the clause (g); if it is a **false** gate, we add the clause $(\neg g)$. If g is a NOT gate, and its predecessor in C is gate h, we add the gates $(\neg g \vee \neg h)$ and $(g \vee h)$ (the conjunctive normal form of $(g \Leftrightarrow \neg h)$). If g is an OR gate with predecessors h and h', then we add to $R(C)$ the clauses $(\neg h \vee g)$, $(\neg h' \vee g)$, and $(h \vee h' \vee \neg g)$ (the conjunctive normal form of $g \Leftrightarrow (h \vee h')$). Similarly, if g is an AND gate with predecessors h and h', then we add to $R(C)$

the clauses $(\neg g \vee h)$, $(\neg g \vee h')$, and $(\neg h \vee \neg h' \vee g)$. Finally, if g is also the output gate, we add to $R(C)$ the clause (g). It is easy to see that $R(C)$ is satisfiable if and only if C was, and that the construction can be carried out within $\log n$ space. □

Example 8.4: One trivial but very useful kind of reduction is *reduction by generalization*. We say, informally, that problem A is a *special case* of problem B if the inputs of A comprise an easily recognizable subset of the inputs of B, and on those inputs A and B have the same answers. For example, CIRCUIT VALUE is a special case of CIRCUIT SAT: Its inputs are all circuits that happen to be variable-free; and on those circuits the CIRCUIT VALUE problem and the CIRCUIT SAT problem have identical answers. Another way to say the same thing is that CIRCUIT SAT is a *generalization* of CIRCUIT VALUE. Notice that there is a trivial reduction from CIRCUIT VALUE to CIRCUIT SAT: Just take R to be the identity function. □

There is a chain of reductions that can be traced in the above examples: From REACHABILITY to CIRCUIT VALUE, to CIRCUIT SAT, to SAT. Do we then have a reduction from REACHABILITY to SAT? That reductions compose requires some proof:

Proposition 8.2: If R is a reduction from language L_1 to L_2 and R' is a reduction from language L_2 to L_3, then the composition $R \cdot R'$ is a reduction from L_1 to L_3.

Proof: That $x \in L_1$ if and only if $R'(R(x)) \in L_3$ is immediate from the fact that R and R' are reductions. The nontrivial part is to show that $R \cdot R'$ can be computed in space $\log n$.

One first idea is to compose the two machines with input and output, M_R and $M_{R'}$, that compute R and R' respectively (Figure 8.2) so that $R(x)$ is first produced, and from it the final output $R'(R(x))$. Alas, the composite machine M must have $R(x)$ written on a work string; and $R(x)$ may be much longer than $\log |x|$.

The solution to this problem is clever and simple: We do not explicitly store the intermediate result in a string of M. Instead, we simulate $M_{R'}$ on input $R(x)$ by remembering at all times the cursor position i of the input string of $M_{R'}$ (which is the output string of M_R). i is stored in binary in a new string of M. Initially $i = 1$, and we have a separate set of strings on which we are about to begin the simulation of M_R on input x.

Since we know that the input cursor in the beginning scans a $\triangleright$, it is easy to simulate the first move of $M_{R'}$. Whenever the cursor of $M_{R'}$'s input string moves to the right, we increment i by one, and continue the computation of machine M_R on input x (on the separate set of strings) long enough for it to produce the next output symbol; this is the symbol currently scanned by the input cursor of $M_{R'}$, and so the simulation can go on. If the cursor stays at the

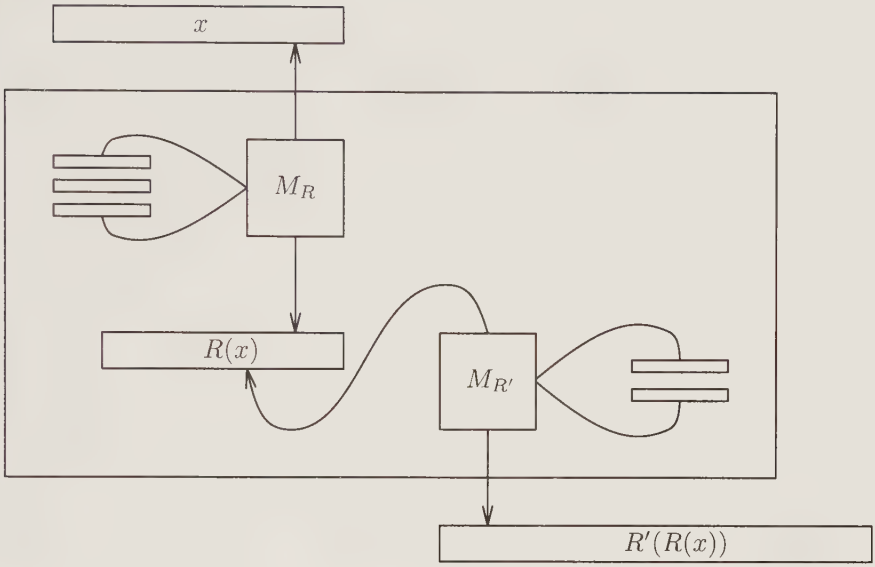


Figure 8-2. How *not* to compose reductions.

same position, we just remember the input symbol scanned. If, however, the input cursor of $M_{R'}$ moves to the left, there is no obvious way to continue the simulation, since the previous symbol output by M_R has been long forgotten. We must do something more radical: We decrement i by one, and then run M_R on x *from the beginning*, counting on a separate string the symbols output, and stopping when the ith symbol is output. Once we know this symbol, the simulation of $M_{R'}$ can be resumed. It is clear that this machine indeed computes $R \cdot R'$ in space $\log n$ (recall that $|R(x)|$ is at most polynomial in $n = |x|$, and so i has $\mathcal{O}(\log n)$ bits). □

8.2 COMPLETENESS

Since reducibility is transitive, it orders problems with respect to their difficulty. We shall be particularly interested in the *maximal elements* of this partial order:

Definition 8.2: Let $\mathcal{C}$ be a complexity class, and let L be a language in $\mathcal{C}$. We say that L is *$\mathcal{C}$-complete* if any language $L' \in \mathcal{C}$ can be reduced to L. □

Although it is not *a priori* clear that complete problems exist, we shall soon show that certain natural and familiar problems are **NP**-complete, and others **P**-complete; in future chapters we shall introduce **PSPACE**-complete problems, **NL**-complete ones, and more.

Complete problems comprise an extremely central concept and method-

ological tool for complexity theory (with **NP**-complete problems perhaps the best-known example). We feel that we have completely understood and categorized the complexity of a problem only if the problem is known to be complete for its complexity class. On the other hand, complete problems capture the essence and difficulty of a class. They are the link that keeps complexity classes alive and anchored in computational practice. For example, the existence of important, natural problems that are complete for a class lends the class a significance that may not be immediately clear from its definition (**NP** is a case in point). Conversely, the absence of natural complete problems makes a class suspect of being artificial and superfluous. However, the most common use of completeness is as a *negative complexity result:* A complete problem is the least likely among all problems in $\mathcal{C}$ to belong in a weaker class $\mathcal{C}' \subseteq \mathcal{C}$; if it does, then the whole class $\mathcal{C}$ coincides with the weaker class $\mathcal{C}'$ *—as long as $\mathcal{C}'$ is closed under reductions.* We say that a class $\mathcal{C}'$ is closed under reductions if, whenever L is reducible to L' and $L' \in \mathcal{C}'$, then also $L \in \mathcal{C}'$. All classes of interest are of this kind:

Proposition 8.3: **P**, **NP**, **coNP**, **L**, **NL**, **PSPACE**, and **EXP** are all closed under reductions.

Proof: Problem 8.4.3. □

Hence, if a **P**-complete problem is in **L**, then **P** = **L**, and if it is in **NL** then **P** = **NL**. If an **NP**-complete problem is in **P**, then **P** = **NP**. And so on. In this sense, *complete problems are a valuable tool for showing complexity classes coincide:*

Proposition 8.4: If two classes $\mathcal{C}$ and $\mathcal{C}'$ are both closed under reductions, and there is a language L which is complete for both $\mathcal{C}$ and $\mathcal{C}'$, then $\mathcal{C} = \mathcal{C}'$.

Proof: Since L is complete for $\mathcal{C}$, all languages in $\mathcal{C}$ reduce to $L \in \mathcal{C}'$. Since $\mathcal{C}'$ is closed under reductions, it follows that $\mathcal{C} \subseteq \mathcal{C}'$. The other inclusion follows by symmetry. □

This result is one aspect of the usefulness of complete problems in the study of complexity. We shall use this method of class identification several times in Chapters 16, 19, and 20.

To exhibit our first **P**-complete and **NP**-complete problems, we employ a useful method for understanding time complexity which could be called the *table method* (recall the reachability method for space complexity). Consider a polynomial-time Turing machine $M = (K, \Sigma, \delta, s)$ deciding language L. Its computation on input x can be thought of as a $|x|^k \times |x|^k$ *computation table* (see Figure 8.3), where $|x|^k$ is the time bound. In this table rows are time steps (ranging from 0 to $|x|^k - 1$), while columns are positions in the string of the machine (the same range). Thus, the (i, j)th table entry represents the contents of position j of the string of M at time i (i.e., after i steps of the machine).

$\triangleright$	0_s	1	1	0	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	1_{q_0}	1	0	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	1	1_{q_0}	0	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	1	1	0_{q_0}	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	1	1	0	$\sqcup_{q_0}$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	1	1	$0_{q'_0}$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	1	1_q	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	1_q	1	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright_q$	1	1	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	1_s	1	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	$\triangleright$	1_{q_1}	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	$\triangleright$	1	$\sqcup_{q_1}$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	$\triangleright$	$1_{q'_1}$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	$\triangleright_q$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	$\triangleright$	$\sqcup_s$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	$\triangleright$	"yes"	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$

Figure 8.3. Computation table.

We shall standardize the computation table a little, so that it is more simple and flexible. Since we know that any k-string Turing machine can be simulated by a single-string machine within a polynomial, it is not a loss of generality to assume that M has one string, and that on any input x it halts after at most $|x|^k - 2$ steps (we can take k high enough so that this holds for $x \geq 2$, and we ignore what happens when $|x| \leq 1$). The computation table pads the string with enough $\sqcup$'s to its right end so that the total length is $|x|^k$; since Turing machine strings are extended during the computation by $\sqcup$'s, this padding of the computation table is no departure from our conventions. Notice that the actual computation will never get to the right end of the table, for lack of time. If at time i the state is q and the cursor scans the jth position, then the (i, j)th entry of the table is not just the symbol σ contained at position j at time i, but a new symbol σ_q, and thus cursor position and state are also recorded nicely. However, if q above is "yes" or "no", then instead of the symbol σ_q we simply have "yes" or "no" as an entry of the table.

We further modify the machine so that the cursor starts not at $\triangleright$, but at the first symbol of the input. Also, the cursor never visits the leftmost $\triangleright$; since such a visit would be followed immediately by a right move, this is achieved by telescoping two moves of the machine each time the cursor is about to move to the leftmost $\triangleright$. Thus, the first symbol in every row of the computation table is a $\triangleright$ (and never an $\triangleright_q$). Finally, we shall assume that, if the machine has halted

before its time bound of n^k has expired, and thus one of the symbols "yes" or "no" appear at a row before the last, then all subsequent rows will be identical to that one. We say that the table is *accepting* if $T_{|x|^k-1,j}$ = "yes" for some j.

Example 8.5: Recall the machine of Example 2.3 deciding palindromes within time n^2. In Figure 8.3 we show its computation table for input 0110. It is an accepting table. □

The folowing result follows immediately from the definition of the computation table:

Proposition 8.5: M accepts x if and only if the computation table of M on input x is accepting. □

We are now ready for our first completeness result:

Theorem 8.1: CIRCUIT VALUE is **P**-complete.

Proof: We know that CIRCUIT VALUE is in **P** (this is a prerequisite for a problem to be **P**-complete, recall Definition 8.2). We shall show that for any language $L \in \mathbf{P}$ there is a reduction R from L to CIRCUIT VALUE.

Given any input x, $R(x)$ must be a variable-free circuit such that $x \in L$ if and only if the value of $R(x)$ is **true**. Let M be the Turing machine that decides L in time n^k, and consider the computation table of M on x, call it T. When $i = 0$, or $j = 0$, or $j = |x|^k - 1$, then the value of T_{ij} is *a priori* known (the jth symbol of x or a $\sqcup$ in the first case, a $\triangleright$ in the second, a $\sqcup$ in the third).

Consider now any other entry T_{ij} of the table. The value of T_{ij} reflects the contents of position j of the string at time i, *which depends only on the contents of the same position or adjacent positions at time* $i - 1$. That is, T_{ij} depends only on the entries $T_{i-1,j-1}$, $T_{i-1,j}$, and $T_{i-1,j+1}$ (see Figure 8.4(a)). For example, if all three entries are symbols in Σ, then this means that the cursor at step i is not at or around position j of the string, and hence T_{ij} is the same as $T_{i-1,j}$. If one of the entries $T_{i-1,j-1}$, $T_{i-1,j}$, or $T_{i-1,j+1}$ is of the form σ_q, then T_{ij} may be a new symbol written at step i, or of the form σ_q if the cursor moves to position j, or perhaps again the same symbol as $T_{i-1,j}$. *In all cases, to determine* T_{ij} *we need only look at* $T_{i-1,j-1}$, $T_{i-1,j}$, *and* $T_{i-1,j+1}$.

Let Γ denote the set of all symbols that can appear on the table (symbols of the alphabet of M, or symbol-state combinations). Encode next each symbol $\sigma \in \Gamma$ as a vector $(s_1, \ldots, s_m)$, where $s_1, \ldots, s_m \in \{0, 1\}$, and $m = \lceil \log |\Gamma| \rceil$. The computation table can now be thought of as a table of binary entries $S_{ij\ell}$ with $0 \leq i \leq n^k - 1$, $0 \leq j \leq n^k - 1$, and $1 \leq \ell \leq m$. By the observation in the previous paragraph, each binary entry $S_{ij\ell}$ only depends on the $3m$ entries $S_{i-1,j-1,\ell'}$, $S_{i-1,j,\ell'}$, and $S_{i-1,j+1,\ell'}$, where ℓ' ranges over $1, \ldots, m$. That is, *there are* m *Boolean functions* $F_1, \ldots, F_m$ with $3m$ inputs each such that, for all $i, j > 0$

$$S_{ij\ell} = F_\ell(S_{i-1,j-1,1}, \ldots, S_{i-1,j-1,m}, S_{i-1,j,1}, \ldots, S_{i-1,j+1,m})$$

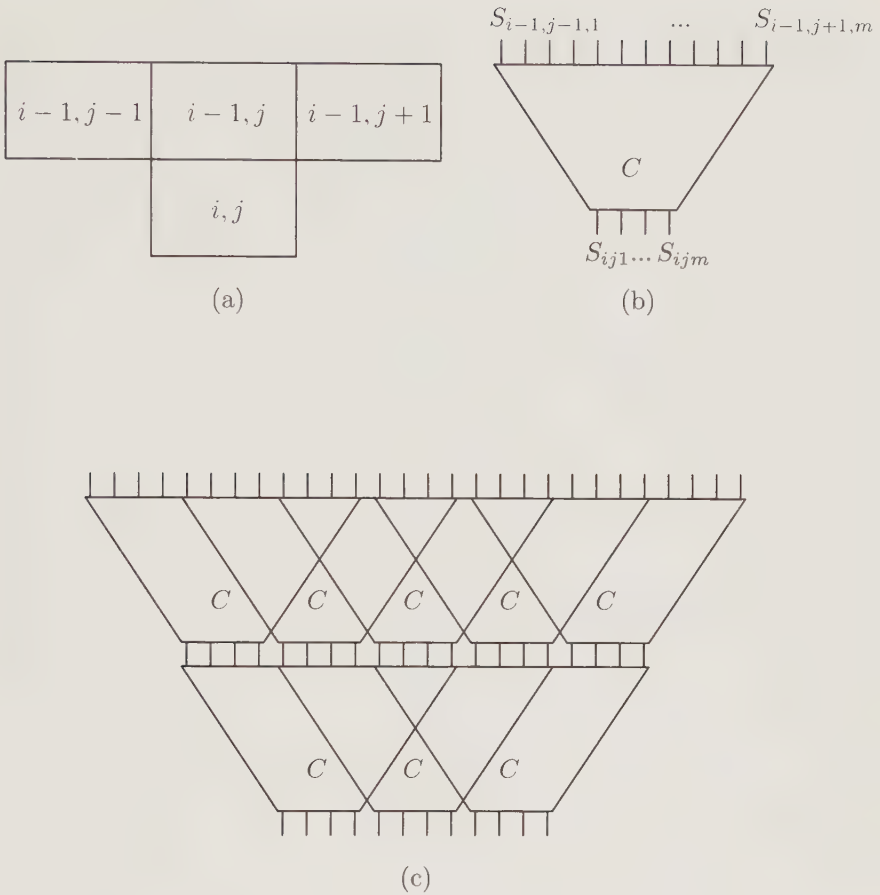


Figure 8-4. The construction of the circuit.

(we call these functions Boolean by disregarding for a moment the difference between **false-true** and 0-1). Since every Boolean function can be rendered as a Boolean circuit (recall Section 4.3), it follows that there is a Boolean circuit C with $3m$ inputs and m outputs that computes the binary encoding of T_{ij} given the binary encodings of $T_{i-1,j-1}$, $T_{i-1,j}$, and $T_{i-1,j+1}$ for all $i = 1, \ldots, |x|^k$ and $j = 1, \ldots, |x|^k - 1$ (see Figure 8.4(b)). Circuit C depends only on M, and has a fixed, constant size, independent of the length of x.

We are now ready to describe our reduction R from L, the language in **P** decided by M, to CIRCUIT VALUE. For each input x, $R(x)$ *will basically consist of* $(|x|^k - 1) \cdot (|x|^k - 2)$ *copies of the circuit* C (Figure 8.4(c)), one for each entry T_{ij} of the computation table that is not on the top row or the two extreme columns. Let us call C_{ij} the (i,j)th copy of C. For $i \geq 1$, the input gates of C_{ij} will be identified with the output gates of $C_{i-1,j-1}$, $C_{i-1,j}$, and $C_{i-1,j+1}$. The input gates of the overall circuit are the gates corresponding to the first row, and the first and last column. The sorts (**true** or **false**) of these gates correspond to the known contents of these three lines. Finally, the output gate of the $R(x)$ is the first output of circuit $C_{|x|^k-1,1}$ (here we are assuming, with no loss of generality, that M always ends with "yes" or "no" on its second string position, and that the first bit of the encoding of "yes" is 1, whereas the of "no" is 0). This completes the description of $R(x)$.

We claim that the value of circuit $R(x)$ is **true** if and only if $x \in L$. Suppose that the value of $R(x)$ is indeed **true**. It is easy to establish by induction on i that the values of the outputs of circuits C_{ij} spell in binary the entries of the computation table of M on x. Since the output of $R(x)$ is **true**, this means that entry $T_{|x|^k-1,1}$ of the computation table is "yes" (since it can only be "yes" or "no", and the encoding of "no" starts with a 0). It follows that the table is accepting, and thus M accepts x; therefore $x \in L$.

Conversely, if $x \in L$ then the computation table is accepting, and thus the value of the circuit $R(x)$ is **true**, as required.

It remains to argue that R can be carried out in $\log |x|$ space. Recall that circuit C is fixed, depending only on M. The computation of R entails constructing the input gates (easy to do by inspecting x and counting up to $|x|^k$), and generating many indexed copies of the fixed circuit C and identifying appropriate input and output gates of these copies— tasks involving straightforward manipulations of indices, and thus easy to perform in $\mathcal{O}(\log |x|)$ space. □

In CIRCUIT VALUE we allow AND, OR, and NOT gates in our circuits (besides the input gates, of course). As it turns out, *NOT gates can be eliminated*, and the problem remains **P**-complete. This is rather surprising, because it is well-known that circuits with only AND and OR gates are less expressive than general circuits: They can only compute *monotone Boolean functions* (recall Problem 4.4.13). Despite the fact that monotone circuits are far less expressive than general circuits, monotone circuits with constant inputs are as difficult to evaluate as general ones. To see this, notice that, given any general circuit we can "move the NOTs downwards," applying De Morgan's Laws at each step (basically, changing all ANDs to ORs and vice-versa) until they are applied to inputs. Then we can simply change $\neg$**true** to **false** (creating a new input gate each time), or vice-versa. This modification of the circuit can obviously be carried out in logarithmic space. We therefore have:

Corollary: MONOTONE CIRCUIT VALUE is **P**-complete.

For other special cases of CIRCUIT VALUE see Problems 8.4.7.

We next prove our first **NP**-completeness result, reusing much of the machinery developed for Theorem 8.1.

Theorem 8.2 (Cook's Theorem): SAT is **NP**-complete.

Proof: The problem is in **NP**: Given a satisfiable Boolean expression, a nondeterministic machine can "guess" the satisfying truth assignment and verify it in polynomial time. Since we know (Example 8.3) that CIRCUIT SAT reduces to SAT, we need to show that all languages in **NP** can be reduced to CIRCUIT SAT.

Let $L \in \mathbf{NP}$. We shall describe a reduction R which for each string x constructs a circuit $R(x)$ (with inputs that can be either variables or constants) such that $x \in L$ if and only if $R(x)$ is satisfiable. Since $L \in \mathbf{NP}$, there is a nondeterministic Turing machine $M = (K, \Sigma, \Delta, s)$ that decides L in time n^k. That is, given a string x, there is an accepting computation (sequence of nondeterministic choices) of M on input x if and only if $x \in L$. We assume that M has a single string; furthermore, we can assume that it has at each step *two nondeterministic choices*. If for some state-symbol combinations there are $m > 2$ choices in Δ, we modify M by adding $m-2$ new states so that the same effect is achieved (see Figure 8.5 for an illustration). If for some combination there is only one choice, we consider that the two choices coincide; and finally if for some state-symbol combination there is no choice in Δ, we add to Δ the choice that changes no state, symbol, or position. So, machine M has exactly two choices for each symbol-state combination. One of these choices is called choice 0 and the other choice 1, so that a sequence of nondeterministic choices is simply a bitstring $(c_1, c_2, \ldots, c_{|x|^k-1}) \in \{0,1\}^{|x|^{k-1}}$.

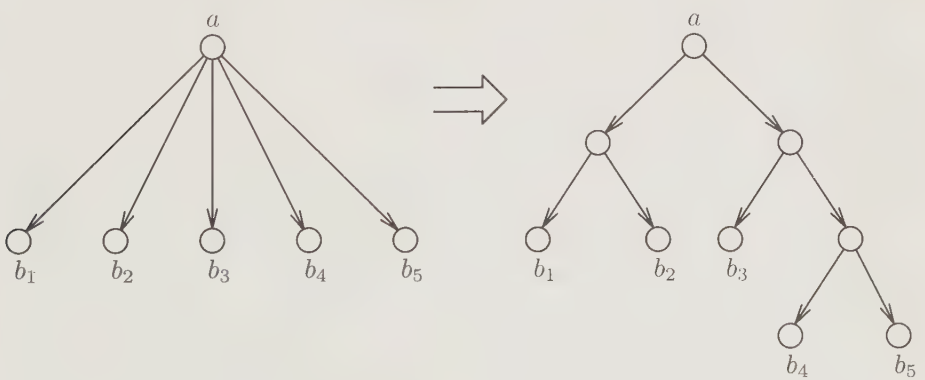


Figure 8-5. Reducing the degree of nondeterminism.

Since the computation of nondeterministic Turing machines proceeds in parallel paths (recall Figure 2.9), there is no simple notion of computation table that captures all of the behavior of such a machine on an input. *If, however, we fix a sequence of choices* $\mathbf{c} = (c_0, c_2, \ldots, c_{|x|^k-1})$, then the computation is effectively deterministic (at the ith step we take choice c_i), and thus we can define the computation table $T(M, x, \mathbf{c})$ corresponding to the machine, input, and sequence of choices. Again the top row and the extreme columns of the table will be predetermined. All other entries T_{ij} will depend only on the entries $T_{i-1,j-1}$, $T_{i-1,j}$, and $T_{i-1,j+1}$ *and the choice* c_{i-1} *at the previous step* (see Figure 8.6). That is, this time the fixed circuit C has $3m + 1$ entries instead of $3m$, the extra entry corresponding to the nondeterministic choice. Thus we can again construct in $\mathcal{O}(\log |x|)$ space a circuit $R(x)$, this time with variable gates $c_0, c_1, \ldots, c_{|x|^k-1}$ corresponding to the nondeterministic choices of the machine. It follows immediately that $R(x)$ is satisfiable (that is, there is a sequence of choices $c_0, c_1, \ldots, c_{|x|^k-1}$ such that the computation table is accepting) if and only if $x \in L$. □

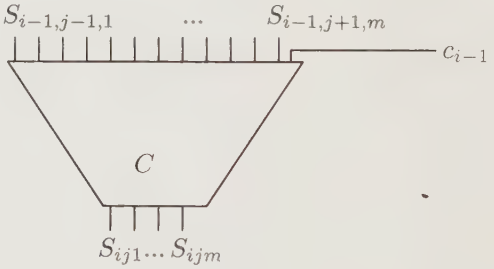


Figure 8-6. The construction for Cook's theorem.

We shall see many more **NP**-complete problems in the next chapter.

8.3 LOGICAL CHARACTERIZATIONS

Theorems 8.1 and 8.2, establishing that two computational problems from logic are complete for our two most important complexity classes, are ample evidence of the close relationship between logic and complexity. There is an interesting parallel, pursued in this section, in which logic captures complexity classes in an even more direct way. Recall that we can associate with each expression ϕ of first-order logic (or of existential second-order logic, recall Section 5.7) in the vocabulary of graph theory a computational problem called ϕ-GRAPHS, asking whether a given finite graph satisfies ϕ. In Section 5.7 we showed that for any expression ϕ in existential second-order logic ϕ-GRAPHS is in **NP**, and it is in **P** if ϕ is Horn. *We shall now show the converse statements.*

Let $\mathcal{G}$ be a set of finite graphs—that is, a graph-theoretic property. The computational problem corresponding to $\mathcal{G}$ is to decide, given a graph G, whether $G \in \mathcal{G}$. We say that $\mathcal{G}$ is *expressible in existential second-order logic* if there is an existential second-order logic sentence $\exists P\phi$ such that $G \models \exists P\phi$ if and only if $G \in \mathcal{G}$.

Naturally, there are many computational problems that do not correspond to properties of graphs. It can be argued, however, that this is an artifact of our preference for strings over graphs as the basis of our encodings. Graphs are perfectly adequate for encoding arbitrary mathematical objects. For example, any language L can be thought of as a set of graphs $\mathcal{G}$, where $G \in \mathcal{G}$ if and only if the first row of the adjacency matrix of G spells a string in L. With this in mind *(and only in this section)* we shall denote by **P** the sets of graph-theoretic properties whose corresponding computational problem is in **P**, and the same for **NP**†.

Theorem 8.3 (Fagin's Theorem): The class of all graph-theoretic properties expressible in existential second-order logic is precisely **NP**.

Proof: If $\mathcal{G}$ is expressible in existential second-order logic, we already know from Theorem 5.8 that it is indeed in **NP**. For the other direction, suppose that $\mathcal{G}$ is a graph property in **NP**. That is, there is a nondeterministic Turing machine M deciding whether $G \in \mathcal{G}$ for some graph G with n nodes in time n^k for some integer $k > 2$. We shall construct a second-order expression $\exists P\phi$ such that $G \models \exists P\phi$ if and only if $G \in \mathcal{G}$.

We must first standardize our nondeterministic machines a little more. We can assume that the input of M is the adjacency matrix of the graph under consideration. In fact, we shall assume that the adjacency matrix is spread over the input string in a rather peculiar way: The input starts off with the $(1,1)$st entry of the adjacency matrix, and between any two entries we have $n^{k-2} - 1$ ⊔'s. That is, the input is spread over n^k positions of the string; since the machine may start by condensing its input, this is no loss of generality.

We are now ready to start our description of $\exists P\phi$. P will be a relation symbol with very high arity. In fact, it will be much more clear to describe an equivalent expression of the form $\exists P_1 \ldots \exists P_m \phi$, where the P_i's are relational symbols; P will then be the Cartesian product of the P_i's. We call the P_i's the *new relations*, and describe them next.

First, S is a binary new relation symbol whose intention is to represent a *successor function over the nodes of G;* that is, in any model M of ϕ, S will be a relation isomorphic to $\{(0,1),(1,2),\ldots,(n-2,n-1)\}$ (notice that here we

† If the reader feels uncomfortable with this sudden change in our most basic conventions, there is another way of stating Theorem 8.3: **NP** is precisely the class of all languages that are *reducible* to some graph-theoretic property expressed in existential second-order logic; similarly for Theorem 8.4. See Problem 8.4.12.

assume that the nodes of G are $0, 1, \ldots, n-1$ instead of the usual $1, 2, \ldots, n$; this is, of course, inconsequential). We shall not describe now how S can be prescribed in first-order logic (Problem 8.4.13); most of the work has already been done in Example 5.12 where a Hamilton path was specified.

Once we have S, and thus we can identify the nodes of G with the integers $0, 1, \ldots, n-1$, we can define some interesting relations. For example, $\zeta(x)$ is an abbreviation of the expression $\forall y \neg S(y, x)$, which states that node x is 0, the element with no predecessor in S; on the other hand, $\eta(j)$ is an expression which abbreviates $\forall y \neg S(x, y)$, stating that node x equals $n-1$.

Since the variables stand for numbers between 0 and $n-1$ (whatever $n-1$ may be in the present model), k-tuples of variables may be used to represent numbers between 0 and $n^k - 1$ *with k the degree of the polynomial bound of M*. We shall abbreviate k-tuples of variables $(x_1, \ldots, x_k)$ as $\mathbf{x}$. In fact, we can define a first-order expression S_k with $2k$ free variables, such that $S_k(\mathbf{x}, \mathbf{y})$ if and only if $\mathbf{y}$ encodes the k-digit n-ary number that comes after the one encoded by $\mathbf{x}$. That is, S_k is the successor function in $\{0, 1, \ldots, n^k - 1\}$.

We shall define S_j inductively on j. First, if $j = 1$ then obviously S_1 is S itself. For the inductive step, suppose that we already have an expression $S_{j-1}(x_1, \ldots, x_{k-1}, y_1, \ldots, y_{j-1})$ that defines the successor function for $j-1$ digits. Then the expression defining S_j is this (universally quantified over all variables):

$$[S(x_j, y_j) \wedge (x_1 = y_1) \wedge \ldots \wedge (x_{j-1} = y_{j-1}))] \vee$$
$$[\eta(x_j) \wedge \zeta(y_j) \wedge S_{j-1}(x_1, \ldots, x_{j-1}, y_1, \ldots, y_{j-1})]$$

That is, in order to obtain from $x_1 \ldots x_j$ the n-ary description of its successor $y_1 \ldots y_j$ (least significant digit first) we do this: If the last digit of $\mathbf{x}$ is not $n-1$ (first line), then we just increment it and keep all other digits the same. But if it is $n-1$, then it becomes zero, and the remaining $j-1$-digit number is incremented, recursively. Thus, $S_k(\mathbf{x}, \mathbf{y})$ is indeed a first-order expression, involving $\mathcal{O}(k^2)$ symbols, satisfied if and only if $\mathbf{x}$ and $\mathbf{y}$ are consecutive integers between zero and $n^k - 1$. It will appear in several places in the expression $\exists P \phi$ described below.

Now that we have S_k, and therefore "we can count up to n^k," we can describe the computation table for M on input x. In particular, for each symbol σ appearing on the computation table, we have a $2k$-ary new relation symbol T_σ. $T_\sigma(\mathbf{x}, \mathbf{y})$ means that the (i, j)th entry of the computation table T is symbol σ, where $\mathbf{x}$ encodes i and $\mathbf{y}$ encodes j. Finally, for the two nondeterministic choices 0 and 1 at each step of M we have two k-ary symbols C_0 and C_1; for example, $C_0(\mathbf{x})$ means that at the ith step, where $\mathbf{x}$ encodes i, the 0th nondeterministic choice is made. These are all the new relations; the second-order formula will thus be of the form $\exists S \exists T_{\sigma_1} \ldots \exists T_{\sigma_k} \exists C_0 \exists C_1 \phi$.

All that remains is to describe ϕ. ϕ essentially states (besides the fact that S is a successor relation, which we omit) the following:

(a) The top row and the extreme columns of T are as they should be in a legal computation table of M on input x.
(b) All remaining entries are filled according to the transition relation of M.
(c) One nondeterministic choice is taken at each step. Finally,
(d) The machine ends accepting.

For part (a) we have to state that, if $\mathbf{x}$ encodes 0 then $T_{\sqcup}(\mathbf{x}, \mathbf{y})$, *unless* the last $k-2$ components of $\mathbf{y}$ are all 0, in which case $T_1(\mathbf{x}, \mathbf{y})$ or $T_0(\mathbf{x}, \mathbf{y})$, *depending on whether or not* $G(y_1, y_2)$, where G is the input graph (recall our peculiar input convention). This is the only place where G occurs in ϕ. Also for part (a) we must state that, if $\mathbf{y}$ encodes 0 then $T_{\triangleright}(\mathbf{x}, \mathbf{y})$, while if $\mathbf{y}$ encodes $n^k - 1$ then $T_{\sqcup}(\mathbf{x}, \mathbf{y})$

For part (b), we must require that the computation table reflects the transition relation of M. Notice that the transition relation of M can be expressed as a set of quintuples $(\alpha, \beta, \gamma, c, \sigma)$ where $\alpha, \beta, \gamma, \sigma$ are table symbols, and $c \in \{0, 1\}$ is a nondeterministic choice. Each such quintuple means that, whenever $T(i-1, j-1) = \alpha$, $T(i-1, j) = \beta$ $T(i-1, j+1) = \gamma$, and the choice c was made at the $i-1$st step, then $T(i, j) = \sigma$. For each such quintuple we have the following conjunct in ϕ:

$$\underline{[S_k(\mathbf{x}', \mathbf{x}) \wedge S_k(\mathbf{y}', \mathbf{y}) \wedge S_k(\mathbf{y}, \mathbf{y}'') \wedge T_\alpha(\mathbf{x}', \mathbf{y}') \wedge T_\beta(\mathbf{x}', \mathbf{y}) \wedge T_\gamma(\mathbf{x}', \mathbf{y}'') \wedge C_c(\mathbf{x}')] \Rightarrow T_\sigma(\mathbf{x}, \mathbf{y})}.$$

The appearances of S_k in this expression are independent copies of the expression defined inductively earlier in the proof.

Part (c) states that at all times exactly one of the nondeterministic choices is taken:

$$(C_0(\mathbf{x}) \vee C_1(\mathbf{x})) \wedge (\neg C_0(\mathbf{x}) \vee \neg C_1(\mathbf{x})). \tag{1}$$

Interestingly, this is a crucial part of the construction — for example, it is the only place where we have a non-Horn clause!

Part (d) is the easiest one: $\theta(\mathbf{x}, \mathbf{y}) \Rightarrow T_{\text{“yes”}}(\mathbf{x}, \mathbf{y})$ where $\theta(\mathbf{x}, \mathbf{y})$ abbreviates the obvious expression stating that $\mathbf{x}$ encodes $n^k - 1$ and $\mathbf{y}$ encodes 1 (recall our convention that the machine stops with “yes” or “no” at the first string position). The conjunction of all these clauses is then preceded by $5k$ universal quantifiers, corresponding to the variable groups $\mathbf{x}$, $\mathbf{x}'$, $\mathbf{y}$, $\mathbf{y}'$, $\mathbf{y}''$. The construction of the expression is now complete.

We claim that a given graph G satisfies the second-order expression above if and only if $G \in \mathcal{G}$. The expression was constructed in such a way that it is satisfied by exactly those graphs G which, when input to M, have an accepting sequence of nondeterministic choices, and thus an accepting computation table; that is, precisely the graphs in $\mathcal{G}$. □

We would have liked to conclude this section with the converse of Theorem 5.9, stating that the set of properties expressible in Horn existential second-order expressions is precisely **P**. Unfortunately, *this is not true.* There are certain computationally trivial graph-theoretic properties such as "the graph has an even number of edges" which cannot be expressed in Horn existential second-order logic (see Problems 8.4.15). The difficulty is of a rather unexpected nature: Of all the ingredients needed to express deterministic polynomial computation (in the style of the previous proof), the only one that cannot be expressed in the Horn fragment is the requirement that S be a successor. *If, however, we augment our logic with a successor relation*, we obtain the desired result.

Let us define precisely what we mean: We say that a graph-theoretic property $\mathcal{G}$ *is expressible in Horn existential second-order logic with successor* if there is a Horn existential second-order expression ϕ with two binary relational symbols G and S, such that the following is true: For any model M appropriate for ϕ *such that* S^M *is a linear order on the nodes of* G^M, $M \models \phi$ if and only if $M^G \in \mathcal{G}$.

Theorem 8.4: The class of all graph-theoretic properties expressible in Horn existential second-order logic with successor is precisely **P**.

Proof: One direction was proven as Theorem 5.9. For the other direction, given a deterministic Turing machine M deciding graph-theoretic property $\mathcal{G}$ within time n^k, we shall construct an expression in Horn existential second-order logic that expresses $\mathcal{G}$ (assuming, of course, that S is a successor). The construction is identical to that of the previous proof, except a little simpler. The constituents of P are now just the T_σ's, since S is now a part of our basic vocabulary. More importantly, there is no C_0 or C_1, since the machine is deterministic. *As result, the expression produced is Horn.* The proof is complete. □

Recall now the special case of SAT with Horn clauses, shown polynomial in Theorem 4.2.

Corollary: HORNSAT is **P**-complete.

Proof: The problem is in **P** by Theorem 4.2. And we know from the proof of Theorem 5.9 that any problem of the form ϕ-GRAPHS, where ϕ is a Horn expression in existential second-order logic, can be reduced to HORNSAT. But Theorem 8.4 says that this accounts for all problems in **P**. □

8.4 NOTES, REFERENCES, AND PROBLEMS

8.4.1 There are many different kinds of reductions; our logarithmic-space reduction is about the weakest kind that has been proposed (and is therefore more useful and convincing as evidence of difficulty); but see Problem 16.4.4 for even weaker reductions, useful for **L** and below. Surprisingly, it is all we need in order to develop the many completeness results in this book –besides, demonstrating that a reduction can be carried out in logarithmic space is usually very easy. Traditionally **NP**-completeness is defined in terms of *polynomial-time many-one reduction* (also called *polynomial transformation*, or *Karp reduction*); and logarithmic-space reductions are only used for **P** and below. It is open whether **NP**-complete problems under the two definitions coincide.

A polynomial-time *Turing reduction* or *Cook reduction* is best explained in terms of oracle machines (see the definition in Section 14.3): Language L Cook-reduces to L' if and only if there is a polynomial-time oracle machine $M^?$ such that $M^{L'}$ decides L. In other words, polynomially many queries of the type "$x \in L'$?" are allowed (and not just one and in the end as with Karp reductions). Polynomial Turing reductions appear to be much stronger (see Section 17.1).

There is an intermediate form of reduction called *polynomial-time truth-table reduction*. In this reduction we can ask several "$x \in L'$?" queries *but all must be asked before any of them is answered*. That is, we obtain the final answer as a Boolean function of the answers (hence the name). For interesting results concerning the four kinds of reductions see

- R. E. Ladner, N. A. Lynch, and A. L. Selman "A comparison of polynomial time reducibilities," *Theor. Comp. Sci., 1*, pp. 103–124, 1975.

But there are many other kinds of reductions: For *nondeterministic reductions* see Problem 10.4.2, and for *randomized reductions* see Section 18.2. In fact, studying the behavior of different kinds of reductions (in their broader sense that includes oracles) and making fine distinctions between them comprises a major part of the research activity in the area known as *structural complexity* (whose annual conference is referenced many times in this book). For a nice exposition of that point of view of complexity see

- J. L. Balcázar, J. Diaz, J. Gabarró *Structural Complexity, vols. I and II*, Springer-Verlag, Berlin, 1988.

Obviously, this direction is rather orthogonal to our concerns here.

8.4.2 Problem: A *linear-time reduction* R must complete its output $R(x)$ in $\mathcal{O}(|x|)$ steps. Prove that there are *no* **P**-complete problems under linear-time reductions. (Such a problem would be in $\mathbf{TIME}(n^k)$ for some fixed $k > 0$.)

8.4.3 Problem: Prove Proposition 8.3, namely that the classes **P**, **NP**, **coNP**, **L**, **NL**, **PSPACE**, and **EXP** are closed under reductions. Is $\mathbf{TIME}(n^2)$ closed under reductions?

8.4.4 Generic complete problems. Show that all languages in $\mathbf{TIME}(f(n))$ reduce to $\{M; x : M \text{ accepts } x \text{ in } f(|x|) \text{ steps}\}$, where $f(n) > n$ is a proper complexity

function. Is this language in $\mathbf{TIME}(f(n))$?

Repeat for nondeterministic classes, and for space complexity classes.

8.4.5 If $\mathcal{C}$ is a complexity class, a language L is called *$\mathcal{C}$-hard* if all languages in $\mathcal{C}$ reduce to L *but L is not known to be in $\mathcal{C}$*. $\mathcal{C}$-hardness implies that L cannot be in any weaker class closed under reductions, unless $\mathcal{C}$ is a subset of that class. But of course L could have much higher complexity than any language in $\mathcal{C}$, and thus it fails to capture the class. For example, many languages decidable in exponential time or worse are trivially **NP**-hard, but they certainly are not as faithful representatives of **NP** as the **NP**-complete problems. We shall not need this concept in this book.

8.4.6 Cook's theorem is of course due to Stephen Cook:

- S. A. Cook "The complexity of theorem-proving procedures," *Proceedings of the 3rd IEEE Symp. on the Foundations of Computer Science*, pp. 151–158, 1971.

A subsequent paper by Richard Karp pointed out the true wealth of **NP**-complete problems (many of these results are proved in the next chapter), and therefore the significance of **NP**-completeness:

- R. M. Karp "Reducibility among combinatorial problems," pp. 85–103 in *Complexity of Computer Computations*, edited by J. W. Thatcher and R. E. Miller, Plenum Press, New York, 1972.

Independently, Leonid Levin showed that several combinatorial problems are "universal for exhaustive search," a concept easily identified with **NP**-completeness (Cook's theorem is sometimes referred to as *the Cook-Levin theorem*).

- L. A. Levin "Universal sorting problems," *Problems of Information Transmission, 9*, pp. 265-266, 1973.

The **P**-completeness of the CIRCUIT VALUE problem (Theorem 8.1) was first pointed out in:

- R. E. Ladner "The circuit value problem is log space complete for P," *SIGACT News, 7*, 1, pp. 18–20, 1975.

8.4.7 Problem: (a) Prove that CIRCUIT VALUE remains **P**-complete even if the circuit is planar. (Show how wires can cross with no harm to the computed value.)

(b) Show that CIRCUIT VALUE can be solved in logarithmic space if the circuit is *both* planar and monotone. (The two parts are from

- L. M. Goldschlager "The monotone and planar circuit value problems are complete for P," *SIGACT News 9*, pp. 25–29, 1977, and
- P. W. Dymond, S. A. Cook "Complexity theory of parallel time and hardware," *Information and Comput., 80*, pp. 205–226, 1989

respectively. So, if your solution of Part (a) did not use NOT gates, maybe you want to check it again...)

8.4.8 Problem: (a) Define a *coding* κ to be a mapping from Σ to Σ, (not necessarily one-to-one). If $x = \sigma_1 \dots \sigma_n \in \Sigma^*$, we define $\kappa(x) = \kappa(\sigma_1) \dots \kappa(\sigma_n)$. Finally, if $L \subseteq \Sigma^*$ is a language, define $\kappa(L) = \{\kappa(x) : x \in L\}$. Show that **NP** is closed under codings.

In contrast, **P** is probably *not* closed under codings, but of course, in view of (a), we cannot prove this without establishing that $\mathbf{P} \neq \mathbf{NP}$. Here is the best we can do:

(b) Show that **P** is closed under codings if and only if $\mathbf{P} = \mathbf{NP}$. (Use SAT.)

8.4.9 Problem: Let $f(n)$ be a function from integers to integers. An *$f(n)$-prover* is an algorithm which, given any valid expression in first-order logic that has a proof in the axiom system in Figure 5.4 of length ℓ, will find this proof in time $f(\ell)$. If the expression is not valid, the algorithm may either report so, or diverge (so the undecidability of validity is not contradicted).

(a) Show that there is a k^n-prover, for some $k > 1$.

In a letter to John von Neumann in 1956, Kurt Gödel hypothesized that an n^k-prover exists, for some $k \geq 1$. For a full translation of this remarkable text, as well as for a discussion of modern-day complexity theory with many interesting historical references, see

- M. Sipser "The history and status of the P versus NP problem," *Proc. of the 24th Annual ACM Symposium on the Theory of Computing,* pp. 603 618, 1992.

Problem: Show that there is an n^k-prover, for some $k \geq 1$, if and only if $\mathbf{P} = \mathbf{NP}$.

8.4.10 Fagin's theorem 8.3 is from

- R. Fagin "Generalized first-order spectra and polynomial-time recognizable sets," pp. 43–73 in *Complexity of Computation*, edited by R. M. Karp, SIAM-AMS Proceedings, vol. 7, 1974.

Theorem 8.4 was implicit independently in

- N. Immerman "Relational queries computable in polynomial time," *Information and Control, 68*, pp. 86–104, 1986;
- M. Y. Vardi "The complexity of relational query languages," *Proceedings of the 14th ACM Symp. on the Theory of Computing*, pp. 137–146, 1962; and
- C. H. Papadimitriou "A note on the expressive power of PROLOG," *Bull. of the EATCS, 26,* pp. 21–23, 1985.

The latter paper emphasizes an interesting interpretation of Theorem 8.3 in terms of the logic programming language PROLOG: Functionless PROLOG programs can decide precisely the languages in **P**. The current statement of Theorem 8.4 is based on

- E. Grädel "The expressive power of second-order Horn logic," *Proc. 8th Symp. on Theor. Aspects of Comp. Sci.*, vol. 480 of Lecture Notes in Computer Science, pp. 466–477, 1991.

8.4.11 Problem: Give an expression in first-order logic describing the successor function S in the proof of Fagin's theorem (Theorem 8.3). (Define a Hamilton path P as in Example 5.12, only without requiring that it be a subgraph of G, and then define a new relation S that omits all transitive edges from P.)

8.4.12 Problem: We can state Fagin's theorem without redefining **NP** as a class of sets of graphs, as follows: **NP** is precisely the class of all languages that are reducible

to a graph-theoretic property which is expressible in existential second-order logic.
(a) Prove this version of Fagin's theorem. (Encode strings as graphs.)
(b) State and prove a similar version of Theorem 8.4.

8.4.13 Problem: Show that **NP** is precisely the set of all graph-theoretic properties which can be expressed in fixpoint logic with successor (recall Problem 5.9.14).

8.4.14 Problem: Sketch a direct proof of Cook's theorem from Fagin's theorem.

8.4.15 It turns out that any graph property ϕ expressible in Horn existential second-order logic obeys a powerful *zero-one law:* If all graphs on n nodes are equiprobable, then the probability that a graph with n nodes satisfies ϕ is either asymptotically zero, or asymptotically one as n goes to infinity; see

- P. Kolaitis and M. Vardi "0 – 1 laws and decision problems for fragments of second-order logic," *Proc. 3rd IEEE Symp. on Logic In Comp. Sci,* pp. 2–11, 1988.

Problem: Based on this result, show that there are trivial properties of graphs, such as the property of having an even number of edges, which are not expressible in Horn existential second-order logic without successor. (What is the probability that a graph has an even number of edges?)

CHAPTER 9

NP-COMPLETE PROBLEMS

Proving NP-completeness results is an important ingredient of our methodology for studying computational problems. It is also something of an art form.

9.1 PROBLEMS IN NP

We have defined **NP** as the class of languages decided by nondeterministic Turing machines in polynomial time. We shall next show an alternative way of looking at **NP**, somewhat akin to its characterization in terms of existential second-order logic (Theorem 8.3). Let $R \subseteq \Sigma^* \times \Sigma^*$ be a binary relation on strings. R is called *polynomially decidable* if there is a deterministic Turing machine deciding the language $\{x; y : (x, y) \in R\}$ in polynomial time. We say that R is *polynomially balanced* if $(x, y) \in R$ implies $|y| \leq |x|^k$ for some $k \geq 1$. That is, the length of the second component is always bounded by a polynomial in the length of the first (the other way is not important).

Proposition 9.1: Let $L \subseteq \Sigma^*$ be a language. $L \in$ **NP** if and only if there is a polynomially decidable and polynomially balanced relation R, such that $L = \{x : (x, y) \in R \text{ for some } y\}$.

Proof: Suppose that such an R exists. Then L is decided by the following nondeterministic machine M: On input x, M guesses a y of length at most $|x|^k$ (the polynomial balance bound for R), and then uses the polynomial algorithm on $x; y$ to test whether $(x, y) \in R$. If so, it accepts, otherwise it rejects. It is immediate that an accepting computation exists if and only if $x \in L$.

Conversely, suppose that $L \in$ **NP**; that is, there is a nondeterministic Turing machine N that decides L in time $|x|^k$, for some k. Define the following relation R: $(x, y) \in R$ if and only if y is the encoding of an accepting

computation of N on input x. It is clear that R is polynomially balanced (since N is polynomially bounded), and polynomially decidable (since it can be checked in linear time whether y indeed encodes an accepting computation of N on x). Furthermore, by our assumption that N decides L, we have that $L = \{x : (x, y) \in R \text{ for some } y\}$. □

Proposition 9.1 is perhaps the most intuitive way of understanding **NP**. Each problem in **NP** has a remarkable property: Any "yes" instance x of the problem has at least one *succinct certificate* (or *polynomial witness*) y of its being a "yes" instance. Naturally, "no" instances possess no such certificates. We may not know how to discover this certificate in polynomial time, but we are sure it exists if the instance is a "yes" instance. For SAT, the certificate of a Boolean expression ϕ is a truth assignment T that satisfies ϕ. T is succinct relative to ϕ (it assigns truth values to variables appearing in ϕ), and it exists if and only if the expression is satisfiable. In HAMILTON PATH, the certificate of a graph G is precisely a Hamilton path of G.

It is now easy to explain why **NP** is inhabited by such a tremendous wealth of practically important, natural computational problems (see the problems mentioned in this chapter, and the references). Many computational problems in several application areas call for the design of mathematical objects of various sorts (paths, truth assignments, solutions of equations, register allocations, traveling salesman routes, VLSI layouts, and so on). Sometimes we seek the optimum among all possible alternatives, and sometimes we are satisfied with any object that fits the design specifications (and we have seen in the example of TSP (D) that optimality can be couched in terms of constraint satisfaction by adding a "budget" to the problem). The object sought is thus the "certificate" that shows the problem is in **NP**. Often certificates are mathematical abstractions of actual, physical objects or real-life plans that will ultimately be constructed or implemented. Hence, it is only natural that in most applications the certificates are not astronomically large, in terms of the input data. And specifications are usually simple, checkable in polynomial time. One should therefore expect that most problems arising in computational practice are in **NP**.

And in fact they are. Although in later chapters we shall see several practically important, natural problems that are not believed to be in **NP**, such problems are not the rule. The study of the complexity of computational problems concerns itself for the most part with specimens in **NP**; it basically tries to sort out which of these problems can be solved in polynomial time, and which cannot. In this context, **NP**-completeness is a most important tool. Showing that the problem being studied is **NP**-complete establishes that it is among the least likely to be in **P**, those that can be solved in polynomial time only if **P** = **NP**. In this sense, **NP**-completeness is a valuable component of our methodology, one that complements algorithm design techniques. An aspect of

its importance is this: Once our problem has been shown **NP**-complete, it seems reasonable† to direct our efforts to the many *alternative approaches* available for such problems: Developing approximation algorithms, attacking special cases, studying the average performance of algorithms, developing randomized algorithms, designing exponential algorithms that are practical for small instances, resorting to local search and other heuristics, and so on. Many of these approaches are integral parts of the theory of algorithms and complexity (see the references and later chapters), and owe their existence and flourish precisely to **NP**-completeness.

9.2 VARIANTS OF SATISFIABILITY

Any computational problem, if generalized enough, will become **NP**-complete or worse. And any problem has special cases that are in **P**. The interesting part is to find the dividing line. SAT provides a very interesting example, pursued in this section.

There are many ways and styles for proving a special case of an **NP**-complete problem to be **NP**-complete. The simplest one (and perhaps the most useful) is when we just have to observe that the reduction we already know creates instances belonging to the special case considered. For example, let kSAT, where $k \geq 1$ is an integer, be the special case of SAT in which the formula is in conjunctive normal form, and all clauses have k literals.

Proposition 9.2: 3SAT is **NP**-complete.

Proof: Just notice that the reduction in Theorem 8.2 and Example 8.3 produces such expressions. Clauses with one or two literals can be made into equivalent clauses with three literals by duplicating a literal in them once or twice (for a direct reduction from SAT to 3SAT see Problem 9.5.2). □

Notice that in our variants of the satisfiability problem we allow repetitions of literals in the clauses. This is reasonable and simplifying, especially since our reductions from these problems do not assume that the literals in a clause are distinct. Naturally enough, 3SAT remains **NP**-complete even if all literals in a clause are required to be distinct (see Problem 9.5.5). In another front, satisfiability remains **NP**-complete even if we also bound the number of *occurrences* of the variables in the expression.

Proposition 9.3: 3SAT remains **NP**-complete even for expressions in which each each variable is restricted to appear at most three times, and each literal at most twice.

Proof: This is a special kind of a reduction, in which we must show that a

† There is nothing wrong with trying to prove that $\mathbf{P} = \mathbf{NP}$ by developing a polynomial-time algorithm for an **NP**-complete problem. The point is that without an **NP**-completeness proof we would be trying the same thing *without knowing it!*

problem remains **NP**-complete even when the instances are somehow restricted. We accomplish this by showing how to rewrite any instance so that the "undesirable features" of the instance (those that are forbidden in the restriction) go away. In the present case, the undesirable features are variables that appear many times. Consider such a variable x, appearing k times in the expression. We replace the first occurrence of x by x_1, the second by x_2, and so on, where $x_1, x_2, \ldots, x_k$ are k new variables. We must now somehow make sure that these k variables take the same truth value. It is easy to see that this is achieved by adding to our expression the clauses $(\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge \ldots \wedge (\neg x_k \vee x_1)$. □

Notice however that, in order to achieve the restrictions of Proposition 9.3, we had to abandon our requirement that all clauses have *exactly* three literals; the reason behind this retreat is spelled out in Problem 9.5.4.

In analyzing the complexity of a problem, we are trying to define the precise boundary between the polynomial and **NP**-complete cases (although we should not be overconfident that such a boundary necessarily exists, see Section 14.1). For SAT this boundary is well-understood, at least along the dimension of literals per clause: We next show that 2SAT is in **P**. (For the boundary in terms of number of occurrences of variables, in the sense of Proposition 9.2, see Problem 9.5.4; the dividing line is again between two and three!)

Let ϕ be an instance of 2SAT, that is, a set of clauses with two literals each. We can define a graph $G(\phi)$ as follows: The vertices of G are the variables of ϕ and their negations; and there is an arc (α, β) if and only if there is a clause $(\neg\alpha \vee \beta)$ (or $(\beta \vee \neg\alpha)$) in ϕ. Intuitively, these edges capture the logical implications $(\Rightarrow)$ of ϕ. As a result, $G(\phi)$ has a curious symmetry: If (α, β) is an edge, then so is $(\neg\beta, \neg\alpha)$; see Figure 9.1 for an example. Paths in $G(\phi)$ are also valid implications (by the transitivity of $\Rightarrow$). We can show the following:

Theorem 9.1: ϕ is unsatisfiable if and only if there is a variable x such that there are paths from x to $\neg x$ and from $\neg x$ to x in $G(\phi)$.

Proof: Suppose that such paths exist, and still ϕ can be satisfied by a truth assignment T. Suppose that $T(x) =$ **true** (a similar argument works when $T(x) =$ **false**). Since there is a path from x to $\neg x$, and $T(x) =$ **true** while $T(\neg x) =$ **false**, there must be an edge (α, β) along this path such that $T(\alpha) =$ **true** and $T(\beta) =$ **false**. However, since (α, β) is an edge of $G(\phi)$, it follows that $(\neg\alpha \vee \beta)$ is a clause of ϕ. This clause is not satisfied by T, a contradiction.

Conversely, suppose that there is no variable with such paths in $G(\phi)$. We are going to construct a satisfying truth assignment, that is, a truth assignment such that no edge of $G(\phi)$ goes from **true** to **false**. We repeat the following step: We pick a node α whose truth value has not yet been defined, and such that there is no path from α to $\neg\alpha$. We consider all nodes reachable from α in $G(\phi)$, and assign them the value **true**. We also assign **false** to the negations of these nodes (the negations correspond to all these nodes from which $\neg\alpha$ is

$$(x_1 \vee x_2) \wedge (x_1 \vee \neg x_3) \wedge (\neg x_1 \vee x_2) \wedge (x_2 \vee x_3)$$

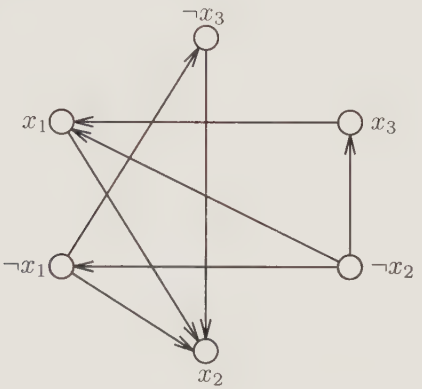


Figure 9-1. The algorithm for 2SAT.

reachable). This step is well-defined because, if there were paths from α to both β and $\neg\beta$, then there would be paths to $\neg\alpha$ from both of these (by the symmetry of $G(\phi)$), and therefore a path from α to $\neg\alpha$, a contradiction of our hypothesis. Furthermore, if there were a path from α to a node already assigned **false** in a previous step, then α is a predecessor of that node, and was also assigned **false** at that step.

We repeat this step until all nodes have a truth assignment. Since we assumed that there are no paths from any x to $\neg x$ and back, all nodes will be assigned a truth value. And since the steps are such that, whenever a node is assigned **true** all of its successors are also assigned **true**, and the opposite for **false**, there can be no edge from **true** to **false**. The truth assignment satisfies ϕ. □

Corollary: 2SAT is in **NL** (and therefore in **P**).

Proof: Since **NL** is closed under complement (Theorem 7.6), we need to to show that we can recognize *unsatisfiable* expressions in **NL**. In nondeterministic logarithmic space we can test the condition of the Theorem by guessing a variable x, and paths from x to $\neg x$ and back. □

A polynomial algorithm, like the one for 2SAT we just described, is not out of place in an **NP**-completeness chapter. Exploring the complexity of a problem typically involves switching back and forth between trying to develop a polynomial algorithm for the problem and trying to prove it **NP**-complete, until one of the approaches succeeds. Incidentally, recall that HORNSAT is another polynomial-time solvable special case of SAT (Theorem 4.2).

It is easy to see that 3SAT is a generalization of 2SAT: 2SAT can be thought of as the special case of 3SAT in which among the three literals in each clause there are at most two distinct ones (remember, we allow repetitions of literals in clauses). We already know that this generalization leads to an **NP**-complete problem. But we can generalize 2SAT in a different direction: 2SAT requires that *all* clauses be satisfied. It would be reasonable to ask whether there is a truth assignment that satisfies not necessarily all clauses, but some large number of them. That is, we are given a set of clauses, each with two literals in it, and an integer K; we are asked whether there is a truth assignment that satisfies at least K of the clauses. We call this problem MAX2SAT; it is obviously an optimization problem, turned into a "yes-no" problem by adding a *goal* K—the counterpart of a budget in maximization problems. MAX2SAT is a generalization of 2SAT, since 2SAT is the special case in which K equals the number of clauses. Notice also that we did not bother to define MAXSAT for clauses with three or more literals, since such a problem would be trivially **NP**-complete—it generalizes the **NP**-complete problem 3SAT.

It turns out that, by generalizing 2SAT to MAX2SAT, we have crossed the **NP**-completeness boundary once more:

Theorem 9.2: MAX2SAT is **NP**-complete.

Proof: Consider the following ten clauses:

$$\begin{array}{c}(x)(y)(z)(w)\\(\neg x \vee \neg y)(\neg y \vee \neg z)(\neg z \vee \neg x)\\(x \vee \neg w)(y \vee \neg w)(z \vee \neg w)\end{array}$$

There is no way to satisfy all these clauses (for example, to satisfy all clauses in the first row we must lose all clauses in the second). But how many *can* we satisfy? Notice first that the clauses are symmetric with respect to x, y, and z (but not w). So, asume that all three of x, y, z are **true**. Then the second row is lost, and we can get all the rest by setting w to **true**. If just two of x, y, z are **true**, then we lose a clause from the first row, and one clause from the second row. Then we have a choice: If we set w to **true**, we get one extra clause from the first row; if we set it to **false**, we get one from the third. So, we can again satisfy seven clauses, and no more. If only one of x, y, z is **true**, then we have one clause from the first row and the whole second row. For the third row, we can satisfy all three clauses by setting w to **false**, but then we lose (w). The maximum is again seven. However, suppose that all three are **false**. Then it is easy to see that we can satisfy at most *six* clauses: The second and third row.

In other words, these ten clauses have the following interesting property: Any truth assignment that satisfies $(x \vee y \vee z)$ can be extended to satisfy seven of them and no more, while the remaining truth assignment can be extended to satisfy only six of them. This suggests an immediate reduction from 3SAT

to MAX2SAT: Given any instance ϕ of 3SAT, we construct an instance $R(\phi)$ of MAX2SAT as follows: For each clause $C_i = (\alpha \vee \beta \vee \gamma)$ of ϕ we add to $R(\phi)$ the ten clauses above, *with* α, β, *and* γ *replacing* x, y, *and* z; w is replaced by a new variable w_i, particular to C_i. We call the ten clauses of $R(\phi)$ corresponding to a clause of ϕ a *group*. If ϕ has m clauses, then obviously $R(\phi)$ has $10m$. The goal is set at $K = 7m$.

We claim that the goal can be achieved in $R(\phi)$ if and only if ϕ is satisfiable. Suppose that $7m$ clauses can be satisfied in $R(\phi)$. Since we know that in each group we can satisfy at most seven clauses, and there are m groups, seven clauses must be satisfied in each group. However, such an assignment would satisfy all clauses in ϕ. Conversely, any assignment that satisfies all clauses in ϕ can be turned into one that satisfies $7m$ clauses in $R(\phi)$ by defining the truth value of w_i in each group according to how many literals of the corresponding clause of ϕ are **true**.

Finally, it is easy to check that MAX2SAT is in **NP**, and that the reduction can be carried out in logarithmic space (since these important prerequisites will be very clear in most of the subsequent reductions, we shall often omit mentioning them explicitly). □

The style of this proof is quite instructive: To show the problem **NP**-complete we start by toying with small instances of the problem, until we isolate one with an interesting behavior (the ten clauses above). Sometimes the properties of this instance immediately enable a simple **NP**-completeness proof. We shall see more uses of this method, sometimes called "gadget construction," in the next section.

We shall end this section with yet another interesting variant of SAT. In 3SAT we are given a set of clauses with three literals in each, and we are asked whether there is a truth assignment T such that no clause has all three literals **false**. Any other combination of truth values in a clause is allowed; in particular, all three literals might very well be **true**. *Suppose now that we disallow this.* That is, we insist that in no clause are all three literals *equal* in truth value (neither all **true**, nor al **false**). We call this problem NAESAT (for "not-all-equal SAT").

Theorem 9.3: NAESAT is **NP**-complete.

Proof: Let us look back at the reduction from CIRCUIT SAT to SAT (Example 8.3). We shall argue that it is essentially also a reduction from CIRCUIT SAT to NAESAT! To see why, consider the clauses created in that reduction. We add to all one- or two-literal clauses among them the same literal, call it z. We claim that the resulting set of clauses, considered as an instance of NAESAT (not 3SAT) is satisfiable if and only if the original circuit is satisfiable.

Suppose that there is a truth assignment T that satisfies all clauses in the sense of NAESAT. It is easy to see that the *complementary* truth assignment $\overline{T}$

also satisfies all clauses in the NAESAT sense. In one of these truth assignments z takes the value **false**. This truth assignment then satisfies all original clauses (before the addition of z), and therefore—by the reduction in Example 8.3—there is a satisfying truth assignment for the circuit.

Conversely, suppose that there is a truth assignment that satisfies the circuit. Then there is a truth assignment T that satisfies all clauses in the ordinary, 3SAT sense; we take $T(z) =$ **false**. We claim that in no clause all literals are **true** under T (we know they are not all **false**). To see why, recall that clauses come in groups corresponding to gates. **true**, **false**, NOT gates and variable gates have clauses involving z, and hence T does not make all of their literals **true**. For an AND gate we have the clauses $(\neg g \vee h \vee z)$, $(\neg g \vee h' \vee z)$, and $(\neg h \vee \neg h' \vee g)$. It is also easy to see that T cannot satisfy all three literals in any clause: This is trivial for the first two clauses since they contain z; and if all literals in the third are **true**, then the other clauses are not satisfied. The case for OR gates is very similar. □

9.3 GRAPH-THEORETIC PROBLEMS

Many interesting graph-theoretic problems are defined in terms of *undirected* graphs. Technically, an undirected graph is just an ordinary graph which happens to be symmetric and have no self-loops; that is, whenever (i, j) is an edge then $i \neq j$, and (j, i) is also an edge. However, since we shall deal with such graphs extensively, we must develop a better notation for them. An undirected graph is a pair $G = (V, E)$, where V is a finite set of nodes and E is a set of *unordered* pairs of nodes in V, called edges; an edge between i and j is denoted $[i, j]$. An edge will be depicted as a line (no arrows). *All graphs in this section are undirected.*

Let $G = (V, E)$ be an undirected graph, and let $I \subseteq V$. We say that the set I is *independent* if whenever $i, j \in I$ then there is no edge between i and j. All graphs (except for the one with no nodes) have non-empty independent sets; the interesting question is, what is the largest independent set in a graph. The INDEPENDENT SET problem is this: Given an undirected graph $G = (V, E)$, and a goal K is there an independent set I with $|I| = K$?.

Theorem 9.4: INDEPENDENT SET is **NP**-complete.

Proof: The proof uses a simple gadget, the *triangle*. The point is that if a graph contains a triangle, then any independent set can obviously contain at most one node of the triangle. But there is more to the construction.

Interestingly, to prove that INDEPENDENT SET is **NP**-complete it is best to *restrict the class of graphs* we consider. Although restricting the domain makes a problem easier, in this case the restriction is such that it retains the complexity of the problem, while making the issues clearer. We consider only graphs *whose nodes can be partitioned in m disjoint triangles (see Figure 9.2).*

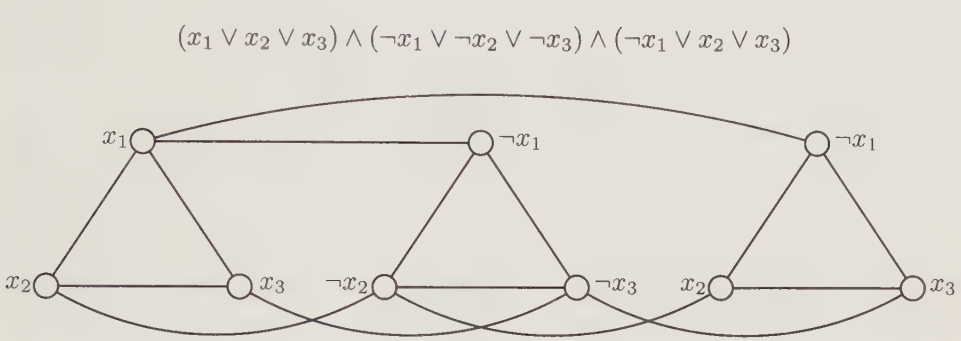


Figure 9-2. Reduction to INDEPENDENT SET.

Then obviously an independent set can contain at most m nodes (one from each triangle); an independent set of size m exists if and only if the other edges of the graph allow us to choose one node from each triangle.

Once we look at such graphs, the combinatorics of the INDEPENDENT SET problem seem easier to understand (and yet no easier to resolve computationally). In fact, a reduction from 3SAT is immediate: For each one of the m clauses of the given expression ϕ we create a separate triangle in the graph G. Each node of the triangle corresponds to a literal in the clause. The structure of ϕ is taken into account in this simple manner: We add an edge between two nodes in different triangles *if and only if the nodes correspond to opposite literals* (see Figure 9.2). The construction is completed by taking the goal to be $K = m$.

Formally, we are given an instance ϕ of 3SAT with m clauses $C_1, \ldots, C_m$, with each clause being $C_i = (\alpha_{i1} \vee \alpha_{i2} \vee \alpha_{i3})$, with the α_{ij}'s being either Boolean variables or negations thereof. Our reduction constructs a graph $R(\phi) = (G, K)$, where $K = m$, and $G = (V, E)$ is the following graph: $V = \{v_{ij} : i = 1, \ldots, m; j = 1, 2, 3\}$; $E = \{[v_{ij}, v_{ik}] : i = 1, \ldots, m; j \neq k\} \cup \{[v_{ij}, v_{\ell k}] : i \neq \ell, \alpha_{ij} = \neg\alpha_{\ell k}\}$. (There is a node for every appearance of a literal in a clause; the first set of edges defines the m triangles, and the second group joins opposing literals.)

We claim that there is an independent set of K nodes in G if and only if ϕ is satisfiable. Suppose that such a set I exists. Since $K = m$, I must contain a node from each triangle. Since the nodes are labeled with literals, and I contains no two nodes corresponding to opposite literals, I is a truth assignment that satisfies ϕ: The **true** literals are just those which are labels of nodes of I (variables left unassigned by this rule can take any value). We know that this gives a truth assignment because any two contradictory literals are connected by an edge in G, and so they cannot both be in I. And since I has a node from every triangle, the truth assignment satisfies all clauses.

Conversely, if a satisfying truth assignment exists, then we identify a **true** literal in each clause, and pick the node in the triangle of this clause labeled by this literal: This way we collect $m = K$ independent nodes. □

By Proposition 9.3, in our proof above we could assume that the original Boolean expression has at most two occurrences of each literal. Thus, each node in the graph constructed in the proof of Theorem 9.4 is adjacent to at most four nodes (that is, it has *degree* at most four): The other two nodes of its triangle, and the two occurrences of the opposite literal. There is a slight complication, because there are clauses now that contain just two literals (recall the proof of Proposition 9.3). But this is easy to fix: Such clauses are represented by a single edge joining the two literals, instead of a triangle. Let k-DEGREE INDEPENDENT SET be the special case of INDEPENDENT SET problem in which all degrees are at most k, an integer; we have shown the following:

Corollary 1: 4-DEGREE INDEPENDENT SET is **NP**-complete. □

Even if the graph is planar, the INDEPENDENT SET problem remains **NP**-complete (see Problem 9.5.9). However, it is polynomially solvable when the graph is *bipartite* (see Problem 9.5.25). The reason for this is that, in bipartite graphs, INDEPENDENT SET is closely related to MATCHING, which in turn is a special case of MAX FLOW.

This brings about an interesting point: Problems in graph theory can be guises of one another in confusing ways; sometimes this suggests trivial reductions from a problem to another. In the CLIQUE problem we are given a graph G and a goal K, and we ask whether there is a set of K nodes that form a *clique* by having all possible edges between them. Also, NODE COVER asks whether there is a set C with B or fewer nodes (where B is a given "budget;" this is a minimization problem) such that each edge of G has at least one of its endpoints in C.

It is easy to see that CLIQUE is a clumsy disguise of INDEPENDENT SET: If we take the *complement* of the graph, that is, a graph that has precisely those edges that are missing from this one, cliques become independent sets and vice-versa. Also, I is an independent set of graph $G = (V, E)$ if and only if $V - I$ is a node cover of the same graph (and, moreover, NODE COVER is, conveniently, a minimization problem). These observations establish the following result:

Corollary 2: CLIQUE and NODE COVER are **NP**-complete. □

A *cut* in an undirected graph $G = (V, E)$ is a partition of the nodes into two nonempty sets S and $V - S$. The *size* of a cut $(S, V - S)$ is the number of edges between S and $V - S$. It is an interesting problem to find the cut with the smallest size in a graph. It turns out that this problem, called MIN CUT, is in **P**. To see why, recall that the smallest cut *that separates two given nodes s and t* equals the maximum flow from s to t (Problem 1.4.11). Thus, to find the minimum overall cut, we just need to find the maximum flow between

some fixed node s and each of the other nodes of V, and pick the smallest value found.

But *maximizing* the size of a cut is much harder:

Theorem 9.5: MAX CUT is **NP**-complete.

Arguably the most important decision in designing an **NP**-completeness proof is, *from which* **NP***-complete problem to start.* Naturally, there are no easy rules here. One should always start by getting as much experience as possible with the problem in hand, examining small and interesting examples. Only then it is perhaps worth going through a (mental or not, see the references) list of known **NP**-complete problems, to see if any problem in there seems very close to the problem in hand. Others always start with 3SAT, an extremely versatile problem which can be easily reduced to a surprising range of **NP**-complete problems (several examples follow). However, in some cases there is tangible dividend from finding the right problem to start: There is a reduction so elegant and simple, that resorting to 3SAT would obviously have been a waste. The following proof is a good example.

Proof: We shall reduce NAESAT to MAX CUT. We are given m clauses with three literals each. We shall construct a graph $G = (V, E)$ and a goal K such that there is a way to separate the nodes of G into two sets S and $V - S$ with K or more edges going from one set to another, if and only if there is a truth assignment that makes at least one literal **true** and at least one literal **false** in each clause. In our construction we shall stretch our definition of a graph by allowing *multiple edges between two nodes;* that is, there may be more than one edge from a node to another, and each of these edges will contribute one to the cut—*if* these nodes are separated.

Suppose that the clauses are $C_1, \ldots, C_m$, and the variables appearing in them $x_1, \ldots, x_n$. G has $2n$ nodes, namely $x_1, \ldots, x_n, \neg x_1, \ldots, \neg x_n$. The gadget employed *is again the triangle*, but this time it is used in a very different way: The property of a triangle we need here is that *the size of its maximum cut is two*, and this is achieved by splitting the nodes in any way. For each clause, say $C_i = (\alpha \vee \beta \vee \gamma)$ (recall that α, β, and γ are also nodes of G), we add to E the three edges of the triangle $[\alpha, \beta, \gamma]$. If two of these literals coincide, we omit the third edge, and the triangle degenerates to two parallel edges between the two distinct literals. Finally, for each variable x_i we add n_i copies of the edge $[x_i, \neg x_i]$, *where n_i is the number of occurrences of x_i or $\neg x_i$ in the clauses.* This completes the construction of G (see Figure 9.3). As for K, it is equal to $5m$.

Suppose that there is a cut $(S, V - S)$ of size $5m$ or more. We claim that it is no loss of generality to assume that all variables are separated from their negations. Because if both x_i and $\neg x_i$ are on the same side of the cut, then together they contribute at most $2n_i$ adjacent edges to the cut, and thus we

$$(x_1 \vee x_2) \wedge (x_1 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \equiv$$
$$(x_1 \vee x_2 \vee x_2) \wedge (x_1 \vee \neg x_3 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3)$$

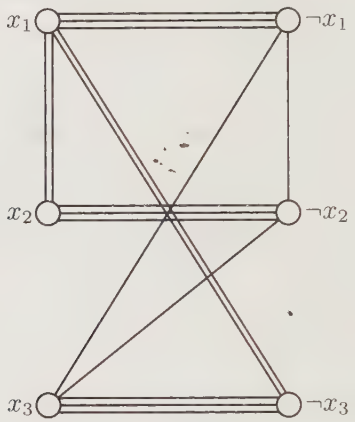


Figure 9-3. Reduction to MAX CUT.

would be able to change the side of one of them without decreasing the size of the cut. Thus, we think of the literals in S as being **true**, and those in $V - S$ as being **false**.

The total number of edges in the cut that join opposite literals is $3m$ (as many as there are occurrences of literals). The remaining $2m$ edges must be obtained from the triangles that correspond to the clauses. And since each triangle can contribute at most two to the size of the cut, all m triangles must be split. However, that a triangle is split means that at least one of its literals is **false**, and at least one **true**; hence, all clauses are satisfied by the truth assignment in the sense of NAESAT.

Conversely, it is easy to translate a truth assignment that satisfies all clauses into a cut of size $5m$. □

In many interesting applications of graph partitioning, the two sets S and $V - S$ cannot be arbitrarily small or large. Suppose that we are looking for a cut $S, V - S$ of size K or more *such that* $|S| = |V - S|$ (if there is an odd number of nodes in V, this problem is *very* easy...). We call this problem MAX BISECTION.

Is MAX BISECTION easier or harder than MAX CUT? Imposing an extra restriction (that $|S| = |V - S|$) certainly makes the problem *conceptually* harder. It also *adversely affects the outcome*, in the sense that the maximum may become smaller. However, an extra requirement could affect the *computational* complexity of the problem both ways. You should have no difficulty recalling (or

devising) problems in **P** for which imposing an extra constraint on the solution space leads to **NP**-completeness, and examples in which exactly the opposite happens. In the case of NAESAT (which results from 3SAT by adding an extra restriction to what it means for a truth assignment to be satisfying) we saw that the problem remains just as hard. This is also the case presently:

Lemma 9.1: MAX BISECTION is **NP**-complete.

Proof: We shall reduce MAX CUT to it. This is a special kind of reduction: We modify the given instance of MAX CUT *so that the extra constraint is easy to satisfy*, and thus the modified instance (of MAX BISECTION) has a solution if and only if the original instance (of MAX CUT) does. The trick here is very simple: *Add $|V|$ completely disconnected new nodes to G.* Since every cut of G can be made into a bisection by appropriately splitting the new nodes between S and $V - S$, the result follows. □

Actually, an even easier way to prove Lemma 9.1 would be to observe that the reduction in the proof of Theorem 9.5 constructs a graph in which the optimum cut *is always a bisection!* How about the minimization version of the bisection problem, called BISECTION WIDTH? It turns out that the extra requirement turns the polynomial problem MIN CUT into an **NP**-complete problem:

Theorem 9.6: BISECTION WIDTH is **NP**-complete.

Proof: Just observe that a graph $G = (V, E)$, where $|V| = 2n$ is an even number, has a bisection of size K or more if and only if the *complement* of G has a bisection of size $n^2 - K$. □

This sequence is instructive for another reason: It touches on the issue of *when a maximization problem is computationally equivalent to the corresponding minimization problem* (see Problem 9.5.14 for some other interesting examples and counterexamples).

We now turn to another genre of graph-theoretic problems. Although HAMILTON PATH was defined for directed graphs, we shall now deal with its undirected special case: Given an undirected graph, does it have a Hamilton path, that is, a path visiting each node exactly once?

Theorem 9.7: HAMILTON PATH is **NP**-complete.

Proof: We shall reduce 3SAT to HAMILTON PATH. We are given a formula ϕ in conjunctive normal form with variables $x_1, \ldots, x_n$ and clauses $C_1, \ldots, C_m$, each with three literals. We shall construct a graph $R(\phi)$ that has a Hamilton path if and only if the formula is satisfiable.

In any reduction from 3SAT we must find ways to express in the domain of the target problem the basic elements of 3SAT; hopefully, everything else will fall in place. But what are the basic ingredients of 3SAT? In an instance of 3SAT we have first of all Boolean variables; the basic attribute of a variable is that it has



Figure 9-4. The choice gadget.

a *choice* between two values, **true** and **false**. We then have occurrences of these variables; the fundamental issue here is *consistency*, that is, all occurrences of x must have the same truth value, and all occurrences of $\neg x$ must have the opposite value. Finally, the occurrences are organized into clauses; it is the clauses that provide the *constraints* that must be satisfied in 3SAT. A typical reduction from 3SAT to a problem constructs an instance that contains parts that "flip-flop" to represent the *choice* by variables; parts of the instance that "propagate" the message of this choice to all occurrences of each variable, thus ensuring *consistency;* and parts that make sure the *constraint* is satisfied. The nature of these parts will vary wildly with the target problem, and ingenuity is sometimes required to take advantage of the intricacies of each specific problem to design the appropriate parts.

In the case of HAMILTON PATH, it is easy to conceive of a *choice* gadget (see Figure 9.4): This simple device will allow the Hamilton path, approaching this subgraph from above, to pick either the left or right parallel edge, thus committing to a truth value. (It will become clear soon that, despite the appearance of Figure 9.4, the graph constructed will have no actual parallel edges.) In this and the other figures in this proof we assume that the devices shown are connected with the rest of the graph only through their *endpoints*, denoted as full dots; there are no edges connecting other nodes of the device to the rest of the graph.

Consistency is ensured using the graph in Figure 9.5(a). Its key property is this: Suppose that this graph is a subgraph of a graph G, connected to the rest of G through its endpoints alone, and suppose that G has a Hamilton path *which does not start or end at a node of this subgraph.* Then this device must be traversed by the Hamilton path in one of two ways, shown in Figures 9.5(b) and (c). To prove this, one has to follow the path as it traverses the subgraph starting from one of the endpoints, and make sure that all deviations from the two ways will lead to a node being left out of the path. This observation

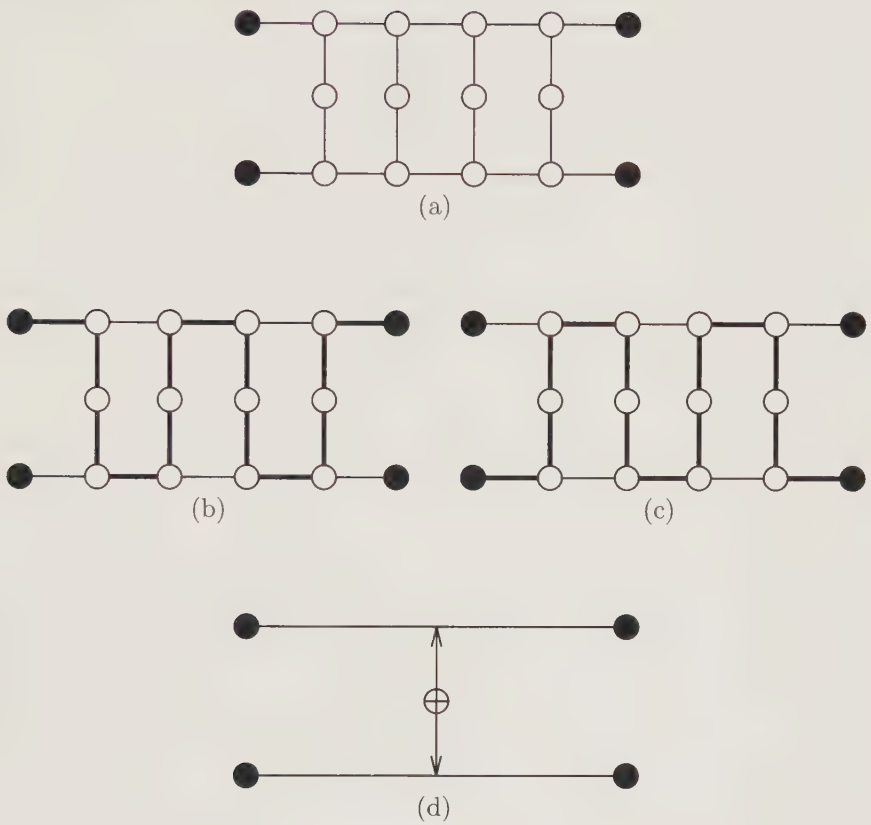


Figure 9-5. The consistency gadget.

establishes that this device behaves as two separate edges, having the property that *in each Hamilton path one of the edges is traversed, and the other is not.* That is, we can think of the device of Figure 9.5(a) as an "exclusive-or" gate connecting two otherwise independent edges of the graph (see Figure 9.5(d)).

How about clauses? How shall we translate the *constraint* imposed by them in the language of Hamilton paths? The device here is, once more, the triangle—one side for each literal in the clause, see Figure 9.6. This is how it works: Suppose that, using our choice and consistency devices, we have made sure that each side of the triangle is traversed by the Hamilton path if and only if the corresponding literal is **false**. Then it is immediate that at least one literal has to be **true**: Otherwise, all three edges of the triangle will be traversed, and hence the alleged "Hamilton path" is neither.

It is now straightforward to put all the pieces together (see Figure 9.7 for

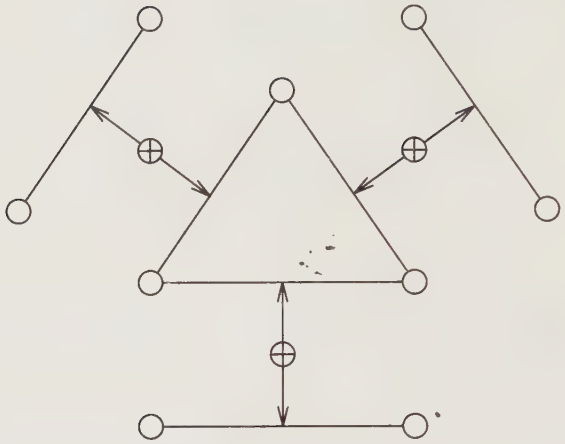


Figure 9-6. The constraint gadget.

an example). Our graph G has n copies of the choice gadget, one for each variable, connected in series (this is the right part of the figure; the first node of the chain is called 1). It also has m triangles, one for each clause, with an edge in the triangle identified with each literal in the clause (in the left part). If a side corresponds to literal x_i, it is connected with an "exclusive or" gadget with the **true** edge of the choice subgraph corresponding to x_i (so that it is traversed if that edge is not); a side corresponding to $\neg x_i$ is connected to the **false** side. (Each **true** or **false** edge may be connected by "exclusive ors" to several sides; these "exclusive ors" are arranged next to each other, as suggested in Figure 9.7.) Finally, all $3m$ nodes of the triangles, plus the last node of the chain of choice gadgets and a new node 3, are connected by all possible edges, creating a huge clique; a single node 2 is attached to node 3 (the last feature only facilitates our proof). This completes our construction of $R(\phi)$.

We claim that the graph has a Hamilton path if and only if ϕ has a satisfying truth assignment. Suppose that a Hamilton path exists. Its two ends must be the two nodes of degree one, 1 and 2, so we can assume that the path starts at node 1 (Figure 9.7). From there, it must traverse one of the two parallel edges of the choice gadget for the first variable. Furthermore, all exclusive ors must be traversed as in Figure 9.5(b) or (c). Therefore the path will continue on after traversing the exclusive ors, and thus the whole chain of choices will be traversed. This part of the Hamilton path defines a truth assignment, call it T. After this, the path will continue to traverse the triangles in some order, and end up at 2.

We claim that T satisfies ϕ. In proof, since all exclusive or gadgets are traversed as in Figure 9.5(b) or (c), they indeed behave like exclusive ors con-

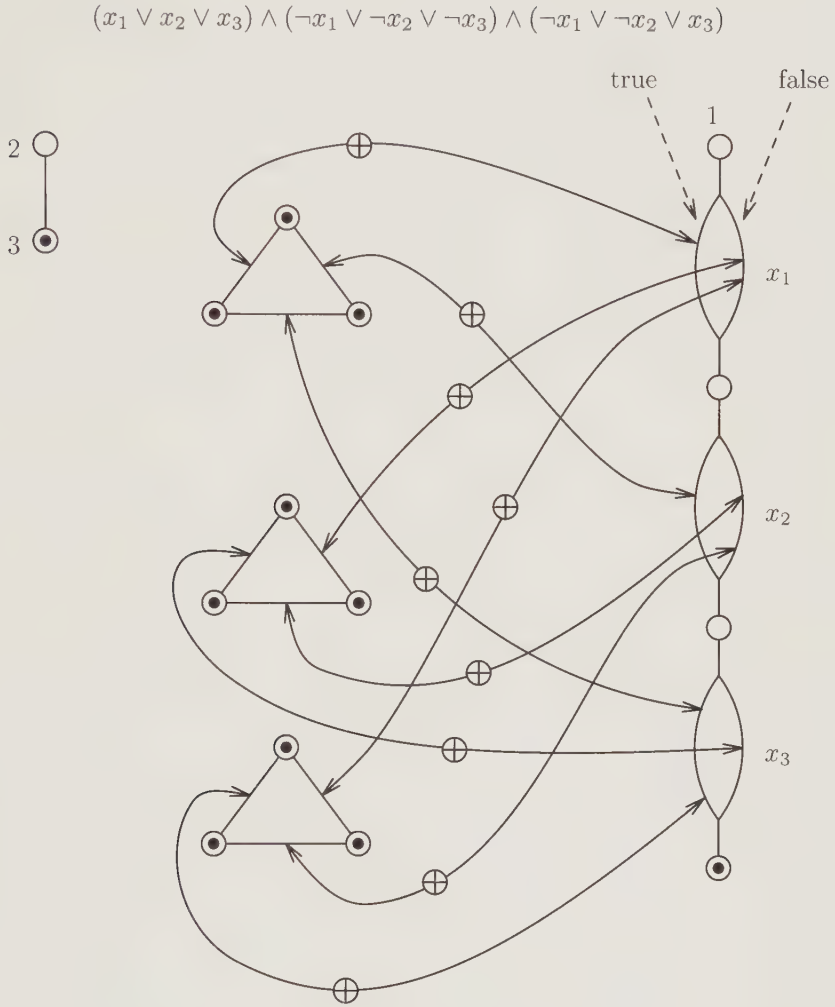


Figure 9-7. The reduction from 3SAT to HAMILTON PATH.

necting otherwise independent edges. So, the sides of triangles corresponding to literals are traversed if and only if the literals are **false**. It follows that there are no clauses that have all three literals **false**, and hence ϕ is satisfied.

Conversely, suppose that there is a truth assignment T that satisfies ϕ.

We shall exhibit a Hamilton path of $R(\varphi)$. The Hamilton path starts at 1, and traverses the chain of choices, picking for each variable the edge corresponding to its truth value under T. Once this is done, the rest of the graph is a huge clique, with certain node-disjoint paths of length two or less that have to be traversed. Since all possible edges are present, it is easy to piece these paths together and complete the Hamilton path, so that it ends at nodes 3 and 2. □

Corollary: TSP (D) is **NP**-complete.

Proof: We shall reduce HAMILTON PATH to it. Given a graph G with n nodes, we shall design a distance matrix d_{ij} and a budget B such that there is a tour of length B or less if and only if G has a Hamilton path. There are n cities, one for each node of the graph. The distance between two cities i and j is 1 if there is an edge $[i, j]$ in G, and 2 otherwise. Finally, $B = n + 1$. The proof that this works is left to the reader. □

Suppose that we are asked to "color" the vertices of a given graph with k colors such that no two adjacent nodes have the same color. This classical problem is called k-COLORING. When $k = 2$ it is quite easy to solve (Problem 1.4.5). For $k = 3$ things change, as usual:

Theorem 9.8: 3-COLORING is **NP**-complete.

Proof: The proof is a simple reduction from NAESAT. We are given a set of clauses $C_1, \ldots, C_m$ each with three literals, involving the variables $x_1, \ldots, x_n$, and we are asked whether there is a truth assignment on the variables such that no clause has all literals **true**, or all literals **false**.

We shall construct a graph G, and argue that it can be colored with colors $\{0, 1, 2\}$ if and only if all clauses can take diverse values. Triangles play an important role again: A triangle forces us to use up all three colors on its nodes. Thus, our graph has for each variable x_i a triangle $[a, x_i, \neg x_i]$; all these triangles share a node a (it is the node at the top colored "2" in Figure 9.8).

Each clause C_i is also represented by a triangle, $[C_{i1}, C_{i2}, C_{i3}]$ (bottom of Figure 9.8). Finally, there is an edge connecting C_{ij} with the node that represents the jth literal of C_i. This completes the construction of the graph G (see Figure 9.8 for an example).

We claim that G can be colored with colors $\{0, 1, 2\}$ if and only if the given instance of NAESAT is satisfiable. In proof, suppose that the graph is indeed 3-colorable. We can assume, by changing color names if necessary, that node a takes the color 2, and so for each i one of the nodes x_i and $\neg x_i$ is colored 1 and the other 0. If x_i takes the color 1 we think that the variable is **true**, otherwise it is **false**. How can the clause triangles be colored? If all literals in a clause are **true**, then the corresponding triangle cannot be colored, since color 1 cannot be used; so the overall graph is not 3-colorable. Similarly if all literal are **false**. This completes the proof of one direction.

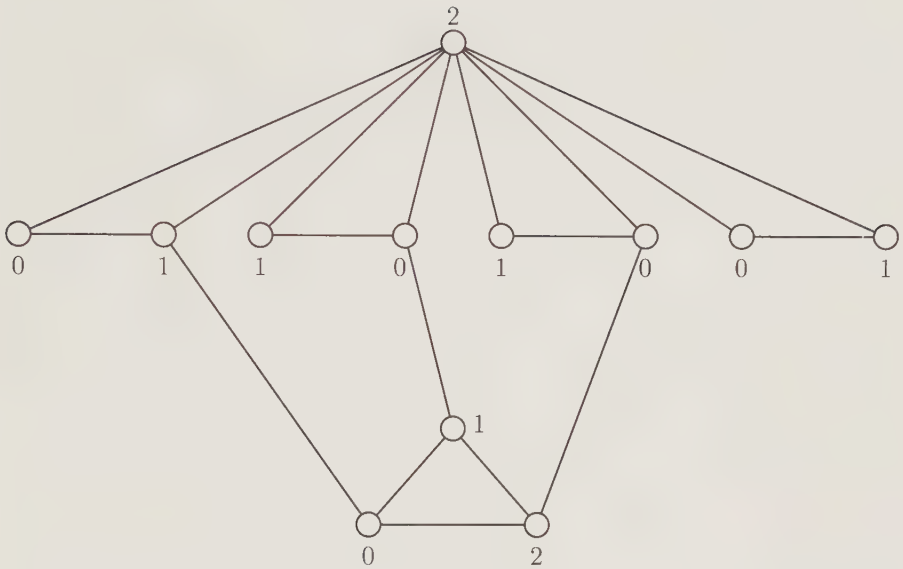


Figure 9-8. The reduction to 3-COLORING.

For the other direction, suppose that a satisfying (in the NAESAT sense) truth assignment exists. We color node a by color 2, and the variable triangles in the way that reflects the truth assignment. And for any clause, we can color the clause triangle as follows: We pick two literals in it with opposite truth values (they exist, since the clause is satisfied) and color the vertices corresponding to them with the available color among $\{0, 1\}$ (0 if the literal is **true**, 1 if it is **false**); we then color the third node 2. □

9.4 SETS AND NUMBERS

We can generalize bipartite matching of Section 1.2 as follows: Suppose that we are given three sets B, G, and H (boys, girls, and *homes*), each containing n elements, and a ternary relation $T \subseteq B \times G \times H$. We are asked to find a set of n triples in T, no two of which have a component in common—that is, each boy is matched to a different girl, and each couple has a home of its own. We call this problem TRIPARTITE MATCHING.

Theorem 9.9: TRIPARTITE MATCHING is **NP**-complete.

Proof: We shall reduce 3SAT to TRIPARTITE MATCHING. The basic ingredient is a *combined gadget for both choice and consistency*, shown in Figure 9.9 (where triples of the relation R are shown as triangles). There is such a device

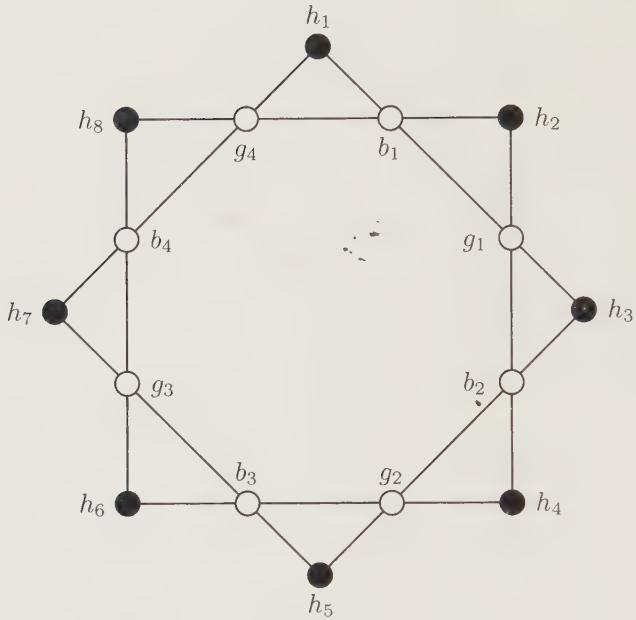


Figure 9-9. The choice-consistency gadget.

for each variable x of the formula. It involves k boys and k girls (forming a circle $2k$ long) and $2k$ homes, where k is either the number of occurrences of x in the formula, or the number of occurrences of $\neg x$, whichever is larger (in the figure, $k = 4$; we could have assumed that $k = 2$, recall Proposition 9.3). Each occurrence of x or $\neg x$ in ϕ is represented by one of the h_j's; however, if x and $\neg x$ have unequal number of occurrences, some of the h_i's will correspond to no occurrence. Homes $h_{2i-1}, i = 1, \ldots, k$ represent occurrences of x, while $h_{2i}, i = 1, \ldots, k$ represent occurrences of $\neg x$. The k boys and k girls participate in no other triple of R other than those shown in the figure. Thus, if a matching exists, b_i is matched either to g_i and h_{2i}, or to g_{i-1} (g_k if $i = 1$) and h_{2i-1}, $i = 1, \ldots, k$. The first matching is taken to mean that $T(x) =$ **true**, the second that $T(x) =$ **false**. Notice that, indeed, this device ensures that variable x picks a truth value, and all of its occurrences have consistent values.

The clause constraint is represented as follows: For each clause c we have a boy and a girl, say b and g. The only triples to which b or g belong are three triples of the form (b, g, h), where h ranges over the three homes corresponding to the three occurrences of the literals in the clause c. The idea is that, if one of these three homes was left unoccupied when the variables were assigned truth values, this means that it corresponds to a **true** literal, and thus c is satisfied.

If all three literals in c are **false**, then b and g cannot be matched with a home.

This would complete the construction, except for one detail: Although the instance has the same number of boys and girls, *there are more homes than either.* If there are m clauses there are going to be $3m$ occurrences, which means that the number of homes, H, is at least $3m$ (for each variable we have at least as many homes as occurrences). On the other hand, there are $\frac{H}{2}$ boys in the choice-consistency gadgets, and $m \leq \frac{H}{3}$ more in the constraint part; so there are indeed fewer boys than homes. Suppose that the excess of homes over boys (and girls) is ℓ—a number easy to calculate from the given instance of 3SAT. We can take care of this problem very easily: We introduce ℓ more boys and ℓ more girls (thus the numbers of boys, girls, and homes are now equal). The ith such girl participates in $|H|$ triples, with the ith boy *and each home.* In other words, these last additions are ℓ "easy to please" couples, useful for completing any matching in which homes were left unoccupied.

We omit the formal proof that a tripartite matching exists if and only if the original Boolean expression was satisfiable. □

There are some other interesting problems involving sets, that we define next. In SET COVERING we are given a family $F = \{S_1, \ldots, S_n\}$ of subsets of a finite set U, and a budget B. We are asking for a set of B sets in F whose union is U. In SET PACKING we are also given a family of subsets of a set U, and a goal K; this time we are asked if there are K pairwise disjoint sets in the family. In a problem called EXACT COVER BY 3-SETS we are given a family $F = \{S_1, \ldots, S_n\}$ of subsets of a set U, such that $|U| = 3m$ for some integer m, and $|S_i| = 3$ for all i. We are asked if there are m sets in F that are disjoint and have U as their union.

We can show all these problems **NP**-complete by pointing out that they are all generalizations of TRIPARTITE MATCHING. This is quite immediate in the case of EXACT COVER BY 3-SETS; TRIPARTITE MATCHING is the special case in which U can be partitioned into three equal sets B, G, and H, such that each set in F contains one element from each. Then, it is easy to see that EXACT COVER BY 3-SETS is the special case of SET COVERING in which the universe has $3m$ elements, all sets in F have three elements, and the budget is m. Similarly for SET PACKING.

Corollary: EXACT COVER BY 3-SETS, SET COVERING, and SET PACKING are **NP**-complete. □

INTEGER PROGRAMMING asks whether a given system of linear inequalities, in n variables and with integer coefficients, has an integer solution. We have already seen more than a dozen reasons why this problem is **NP**-complete: All problems we have seen so far can be easily expressed in terms of linear inequalities over the integers. For example, SET COVERING can be expressed by the inequalities $Ax \geq \mathbf{1}$; $\sum_{i=1}^{n} x_i \leq B$; $0 \leq x_i \leq 1$, where each x_i is a $0 - 1$

variable which is one if and only if S_i is in the cover, A is the matrix whose rows are the bit vectors of the sets, $\mathbf{1}$ is the column vector with all entries 1, and B is the budget of the instance. Hence INTEGER PROGRAMMING is **NP**-complete (the hard part is showing that it is in **NP**; see the notes at the end of the chapter). In contrast, LINEAR PROGRAMMING, the same problem without the requirement that the solutions be integers, is in **P** (see the discussion in 9.5.34).

We shall next look at a very special case of INTEGER PROGRAMMING. The *knapsack problem* looks at the following situation. We must select some among a set of n items. Item i has *value* v_i, and *weight* w_i, both positive integers. There is a limit W to the total weight of the items we can pick. We wish to pick certain items (without repetitions) to maximize the total value, subject to the constraint that the total weight is at most G. That is, we are looking for a subset $S \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in S} w_i \leq W$, and $\sum_{i \in S} v_i$ is as large as possible. In the recognition version, called KNAPSACK, we are also given a goal K, and we wish to find a subset $S \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in S} w_i \leq W$ and $\sum_{i \in S} v_i \geq K$.

→	0	0	0	1	0	1	1	0	0	0	0	0
	1	1	0	0	1	0	0	0	0	0	0	0
→	1	0	1	0	0	0	0	1	0	0	0	0
→	0	1	0	0	0	0	0	0	1	0	0	1
	0	0	1	1	1	0	0	0	0	0	0	0
→	0	0	0	0	1	0	0	0	0	1	1	0
+	1	0	1	1	0	0	0	0	0	0	0	0
	1	1	1	1	1	1	1	1	1	1	1	1

Figure 9.10. Reduction to KNAPSACK.

Theorem 9.10: KNAPSACK is **NP**-complete.

Proof: This is another case in which restricting the problem facilitates the **NP**-completeness proof. We are going to look at the special case of KNAPSACK in which $v_i = w_i$ for all i, and $K = W$. That is, we are given a set of n integers $w_1, \ldots, w_n$, and another integer K, and we wish to find out if a subset of the given integers adds up to exactly K. This simple numerical problem turns out to be **NP**-complete.

We shall reduce EXACT COVER BY 3-SETS to it. We are given an instance $\{S_1, S_2, \ldots, S_n\}$ of EXACT COVER BY 3-SETS, where we are asking whether there are disjoint sets among the given ones that cover the set $U = \{1, 2, \ldots, 3m\}$. Think of the given sets as bit vectors in $\{0,1\}^{3m}$. Such vectors can also be thought as binary integers, and set union now resembles integer addition (see Figure 9.10). Our goal is to find a subset of these integers that add up to

$K = 2^n - 1$ (the all-ones vector, corresponding to the universe). The reduction seems complete!

But there is a "bug" in this reduction: Binary integer addition is different from set union in that it has *carry*. For example, $3 + 5 + 7 = 15$ in bit-vector form is $0011 + 0101 + 0111 = 1111$; but the corresponding sets $\{3, 4\}$,$\{2, 4\}$, and $\{2, 3, 4\}$ are not disjoint, neither is their union $\{1, 2, 3, 4\}$. There is a simple and clever way around this problem: Think of these vectors as integers not in base 2, *but in base* $n + 1$. That is, set S_i becomes integer $w_i = \sum_{j \in S_i} (n+1)^{3m-j}$. Since now there can be no carry in any addition of up to n of these numbers, it is straightforward to argue that there is a set of these integers that adds up to $K = \sum_{j=0}^{3m-1} (n+1)^j$ if and only if there is an exact cover among $\{S_1, S_2, \ldots, S_n\}$. □

Pseudopolynomial Algorithms and Strong NP-completeness

In view of Theorem 9.10, the following result seems rather intriguing:

Proposition 9.4: Any instance of KNAPSACK can be solved in $\mathcal{O}(nW)$ time, where n is the number of items and W is the weight limit.

Proof: Define $V(w, i)$ to be the largest value attainable by selecting some among the i first items *so that their total weight is exactly* w. It is easy to see that the nW entries of the $V(w, i)$ table can be computed in order of increasing i, and with a constant number of operations per entry, as follows:

$$V(w, i+1) = \max\{V(w, i), v_{i+1} + V(w - w_{i+1}, i)\}$$

To start, $V(w, 0) = 0$ for all w. Finally, the given instance of KNAPSACK is a "yes" instance if and only if the table contains an entry greater than or equal to the goal K. □

Naturally, Proposition 9.4 does not establish that **P** = **NP** (so, keep on reading this book!). This is not a polynomial algorithm because its time bound nW is *not a polynomial function of the input:* The length of the input is something like $n \log W$. We have seen this pattern before in our first attempt at an algorithm for MAX FLOW in Section 1.2, when the time required was again polynomial in the integers appearing in the input (instead of their logarithms, which is always the correct measure). Such "pseudopolynomial" algorithms are a source not only of confusion, but of genuinely positive results (see Chapter 13 on approximation algorithms).

In relation to pseudopolynomial algorithms, it is interesting to make the following important distinction between KNAPSACK and the other problems that we showed **NP**-complete in this chapter—SAT, MAX CUT, TSP (D), CLIQUE, TRIPARTITE MATCHING, HAMILTON PATH, and many others. All these latter problems were shown **NP**-complete via reductions that constructed only *polynomially small integers.* For problems such as CLIQUE and SAT, in which integers

are only used as node names and variable indices, this is immediate. But even for TSP (D), in which one would expect numbers to play an important role as intercity distances, we only needed distances no larger than *two* to establish **NP**-completeness (recall the proof of the Corollary to Theorem 9.7)†. In contrast, in our **NP**-completeness proof for KNAPSACK we had to create *exponentially large integers* in our reduction.

If a problem remains **NP**-complete even if any instance of length n is restricted to contain integers of size at most $p(n)$, a polynomial, then we say that the problem is *strongly* **NP**-*complete*. All **NP**-complete problems that we have seen so far in this chapter, with the single exception of KNAPSACK, are strongly **NP**-complete. It is no coincidence then that, of all these problems, only KNAPSACK can be solved by a pseudopolynomial algorithm: It should be clear that strongly **NP**-complete problems *have no pseudopolynomial algorithms*, unless of course $\mathbf{P} = \mathbf{NP}$ (see Problem 9.5.31).

We end this chapter with a last interesting example: A problem which involves numbers and bearing a certain similarity to KNAPSACK, but turns out to be strongly **NP**-complete.

BIN PACKING: We are given N positive integers $a_1, a_2, \ldots, a_N$ (the *items*), and two more integers C (the *capacity*) and B (the *number of bins*). We are asked whether these numbers can be partitioned into B subsets, each of which has total sum at most C.

Theorem 9.11: BIN PACKING is **NP**-complete.

Proof: We shall reduce TRIPARTITE MATCHING to it. We are given a set of boys $B = \{b_1, b_2, \ldots, b_n\}$, a set of girls $G = \{g_1, g_2, \ldots, g_n\}$, a set of homes $H = \{h_1, h_2, \ldots, h_n\}$, and a set of triples $T = \{t_1, \ldots, t_m\} \subseteq B \times G \times H$; we are asked whether there is a set of n triples in T, such that each boy, girl, and home is contained in one of the n triples.

The instance of BIN PACKING that we construct has $N = 4m$ items—one for each triple, and one for each occurrence of a boy, girl, or home to a triple. The items corresponding to the occurrences of b_1, for example, will be denoted by $b_1[1], b_1[2], \ldots, b_1[N(b_1)]$, where $N(b_1)$ is the number of occurrences of b_1 in the triples; similarly for the other boys, the girls, and the homes. The items corresponding to triples will be denoted simply t_j.

The sizes of these items are shown in Figure 9.11. M is a very large number, say $100n$. Notice that one of the occurrences of each boy, girl, and home (arbitrarily the first) has different size than the rest; it is this occurrence that will participate in the matching. The capacity C of each bin is $40M^4 + 15 -$

† To put it otherwise, these problems would remain **NP**-complete if numbers in their instances were represented *in unary*—even such wasteful representation would increase the size of the instance by a only polynomial amount, and thus the reduction would still be a valid one.

just enough to fit a triple and one occurrence of each of its three members as *long as either all three or none of the three are a first occurrence.* There are m bins, as many as triples.

Item	Size
first occurrence of a boy $b_i[1]$	$10M^4 + iM + 1$
other occurrences of a boy $b_i[q], q > 1$	$11M^4 + iM + 1$
first occurrence of a girl $g_j[1]$	$10M^4 + jM^2 + 2$
other occurrences of a girl $g_j[q], q > 1$	$11M^4 + jM^2 + 2$
first occurrence of a home $h_k[1]$	$10M^4 + kM^3 + 4$
other occurrences of a home $h_k[q], q > 1$	$8M^4 + kM^3 + 4$
triple $(b_i, g_j, h_k) \in T$	$10M^4 + 8 - iM - jM^2 - kM^3$

Figure 9.11. The items in BIN PACKING.

Suppose that there is a way to fit these items into m bins. Notice immediately that the sum of all items is mC (the total capacity of all bins), and thus all bins must be exactly full. Consider one bin. It must contain four items (proof: all item sizes are between $\frac{1}{5}$ and $\frac{1}{3}$ of the bin capacity). Since the sum of the items modulo M must be 15 ($C \bmod M = 15$), and there is only one way of creating 15 by choosing four numbers (with repetitions allowed) out of 1, 2, 4, and 8 (these are the residues of all item sizes, see Figure 9.11), the bin must contain a triple that contributes 8 mod M, say (b_i, g_j, h_k), and occurrences of a boy $b_{i'}$, a girl $g_{j'}$, and a home $h_{k'}$, contributing 1, 2, and 4 mod 15, respectively. Since the sum modulo M^2 must be 15 as well, we must have $(i' - i) \cdot M + 15 = 15 \bmod M^2$, and thus $i = i'$. Similarly, taking the sum modulo M^3 we get $j = j'$, and modulo M^4 we get $k = k'$. Thus, each bin contains a triple $t = (b_i, g_j, h_k)$, together with one occurrence of b_i, one of g_j, and one of h_k. Furthermore, either all three occurrences are first occurrences, or none of them are –otherwise $40M^4$ cannot be achieved. Hence, there are n bins that contain only first occurrences; *the n triples in these bins form a tripartite matching.*

Conversely, if a tripartite matching exists, we can fit all items into the m bins by matching each triple with occurrences of its members, making sure that the triples in the matching get first occurrences of all three members. The proof is complete. □

Notice that the numbers constructed in this reduction are polynomially large —$\mathcal{O}(|x|^4)$, where x is the original instance of TRIPARTITE MATCHING. Hence, BIN PACKING is strongly **NP**-complete: Any pseudopolynomial algo-

rithm for BIN PACKING would yield a polynomial algorithm for TRIPARTITE MATCHING, implying $\mathbf{P} = \mathbf{NP}$.

BIN PACKING is a useful point of departure for reductions to problems in which numbers appear to play a central role, but which, unlike KNAPSACK (at least as far as we know), are strongly **NP**-complete.

9.5 NOTES, REFERENCES, AND PROBLEMS

9.5.1 Many of the **NP**-completeness results in this chapter were proved in

- R. M. Karp "Reducibility among combinatorial problems," pp. 85–103 in *Complexity of Computer Computations*, edited by J. W. Thatcher and R. E. Miller, Plenum Press, New York, 1972,

a most influential paper in which the true scope of **NP**-completeness (as well as its importance for combinatorial optimization) was revealed. One can find there early proofs of Theorems 9.3, 9.7, 9.8, and 9.9, as well as the **NP**-completeness results in Problems 9.5.7 and 9.5.12. The definitive work on **NP**-completeness is

- M. R. Garey and D. S. Johnson *Computers and Intractability: A Guide to the Theory of NP-completeness*, Freeman, San Francisco, 1979.

This book contains a list of several hundreds of **NP**-complete problems, *circa* 1979, and is still a most useful reference on the subject (and a good book to leaf through during spells of **NP**-completeness prover's block). David Johnson's continuing commentary on **NP**-completeness, and complexity in general, is a good supplementary reference:

- D. S. Johnson "The NP-completeness column: An on-going guide," *J. of Algorithms, 4*, 1981, and hence.

Proposition 9.1 on the characterization of **NP** in terms of "certificates" was implicit in the work of Jack Edmonds (see the references in 1.4.7).

9.5.2 Problem: Give a direct reduction from SAT to 3SAT. (That is, given a clause with more than three literals, show how to rewrite it as an equivalent set of three-literal clauses, perhaps using additional auxiliary variables.)

9.5.3 Problem: Show that the version of 3SAT in which we require that *exactly one* literal in each clause be true (instead of at least one), called ONE-IN-THREE SAT, is **NP**-complete. (In fact, it remains **NP**-complete even if all literals are positive; see the Corollary to Theorem 9.9 on EXACT COVER BY 3-SETS.)

9.5.4 Problem: (a) Show that the special case of SAT in which each variable can only appear *twice* is in **P**.

In fact, something stronger is true:

(b) Show that the restriction of 3SAT (exactly three literals per clause, no repetitions) in which each variable can appear at most three times, is in **P**. (Consider the bipartite graph with clauses as boys and variables as girls, and edges denoting appearances. Use Problem 9.5.25 to show that it always has a matching.)

9.5.5 Problem: Show that the version of 3SAT in which each clause contains appearances of three *distinct* variables is also **NP**-complete. What is the minimum number of occurrences of variables for which the result holds?

9.5.6 Problem: Show that the special case of SAT in which each clause is *either Horn or has two literals* is **NP**-complete. (In other words, the polynomial special cases of SAT do not mix well.)

9.5.7 Problem: In the DOMINATING SET problem we are given a *directed* graph $G = (V, E)$ and an integer K. We are asking whether there is a set D of K or fewer nodes such that for each $v \notin D$ there is a $u \in D$ with $(u, v) \in E$. Show that DOMINATING SET is **NP**-complete. (Start from NODE COVER, obviously a very similar problem, and make a simple local replacement.)

9.5.8 Problem: A *tournament* is a directed graph such that for all nodes $u \neq v$ exactly one of the edges (u, v) and (u, v) is present. (Why is it called this?)

(a) Show that *every tournament with n nodes has a dominating set of size* $\log n$. (Show that in any tournament there is a player who beats at least half of her opponents; add this player to the dominating set. What is left to dominate?)

(b) Show that, if the DOMINATING SET problem remains **NP**-complete even if the graph is a tournament, then $\mathbf{NP} \subseteq \mathbf{TIME}(n^{k \cdot \log n})$.

The DOMINATING SET problem for tournaments is one of a few natural problems that exhibit a strange sort of "limited nondeterminism;" see

- N. Megiddo and U. Vishkin "On finding a minimum dominating set in a tournament," *Theor. Comp. Sci, 61*, pp. 307–316, 1988, and
- C. H. Papadimitriou and M. Yannakakis "On limited nondeterminism and the complexity of computing the V-C dimension," *Proc. of the 1993 Symposium on Structure in Complexity Theory*.

9.5.9 Problem: (a) Show that the INDEPENDENT SET problem remains **NP**-complete even if the graph is planar. (Show how to replace a crossing of edges so that the basic features of the problem are preserved. Alternatively, see Problem 9.5.16.)

(b) Which of these problems, closely related to INDEPENDENT SET, also remain **NP**-complete in the planar special case? (i) NODE COVER, (ii) CLIQUE, (iii) DOMINATING SET.

9.5.10 Problem: Let $G = (V, E)$ be a directed graph. A *kernel* of G is a subset K of V such that (1) for any two $u, v \in K$ $(u, v) \notin E$, and (2) for any $v \notin K$ there is a $u \in K$ such that $(u, v) \in E$. In other words, a kernel is an independent dominating set.

(a) Show that it is **NP**-complete to tell whether a graph has a kernel. (Two nodes with both arcs between them and no other incoming arcs can act as a flip-flop.)

(b) Show that a strongly connected directed graph with no odd cycles is bipartite (its nodes can be partitioned into two sets, with no edges within the sets), and so it has at least two kernels.

(c) Show that it is **NP**-complete to tell whether a strongly connected directed graph with no odd cycles has a third kernel (besides the two discovered in (b) above).

(d) Based on (b) above show that any directed graph with no odd cycles has a kernel.

(e) Modify the search algorithm in Section 1.1 to decide in polynomial time whether a given directed graph has no odd cycles.

(f) Show that any *symmetric* graph without self-loops (that is, any undirected graph) has a kernel.

(g) Show that the MINIMUM UNDIRECTED KERNEL problem, telling whether an undirected graph has a kernel with at most B nodes, is **NP**-complete. (Similar reduction as in (a).)

9.5.11 Problem: Show that MAX BISECTION remains **NP**-complete even if the given graph is connected.

9.5.12 Problem: In the STEINER TREE problem we are given distances $d_{ij} \geq 0$ between n cities, and a set $M \subseteq \{1, 2, \ldots, n\}$ of *mandatory cities*. Find the shortest possible connected graph that contains the mandatory cities. Show that STEINER TREE is **NP**-complete.

- M. R. Garey, R. L. Graham, and D. S. Johnson "The complexity of computing Steiner minimal trees," *SIAM J. Applied Math., 34,*, pp. 477–495, 1977

show that the EUCLIDEAN STEINER TREE problem (the mandatory nodes are points in the plane, the distances are the Euclidean metric, and all points in the plane are potential non-mandatory nodes; this is the original fascinating problem proposed by Georg Steiner, of which the graph-theoretic version STEINER TREE is a rather uninteresting generalization) is **NP**-complete.

9.5.13 Problem: MINIMUM SPANNING TREE is basically the STEINER TREE problem with all nodes mandatory. We are given cities and nonnegative distances, and we are asked to construct the shortest graph that connects all cities. Show that the optimum graph will have no cycles (hence the name). Show that the *greedy algorithm* (add to the graph the shortest distance not yet considered, unless it is superfluous presumably because its endpoints are already connected) solves this problem in polynomial time.

9.5.14 Problem: Consider the following pairs of minimization-maximization problems in weighted graphs:

(a) MINIMUM SPANNING TREE and MAXIMUM SPANNING TREE (we seek the *heaviest* connecting tree).

(b) SHORTEST PATH (recall Problem 1.4.15) and LONGEST PATH (we seek the longest path with no repeating nodes; this is sometimes called the TAXICAB RIPOFF problem).

(c) MIN CUT between s and t, and MAX CUT between s and t.

(d) MAX WEIGHT COMPLETE MATCHING (in a bipartite graph with edge weights), and MIN WEIGHT COMPLETE MATCHING.

(e) TSP, and the version in which the *longest* tour must be found.

Which of these pairs are polynomially equivalent and which are not? Why?

9.5.15 Problem: (a) Show that the HAMILTON CYCLE problem (we are now seeking a cycle that visits all nodes once) is **NP**-complete. (Modify the proof of Theorem 9.7 a little.)

(b) Show that HAMILTON CYCLE remains **NP**-complete even if the graphs are restricted to be planar, bipartite, and cubic (all degrees three). (Start with the previous reduction, and patiently remove all violations of the three conditions. For planarity,

notice that only exclusive-or gadgets intersect; find a way to have two of them "cross over" with no harm to their function. Alternatively, see Problem 9.5.16.)

(c) A *grid graph* is a finite, induced subgraph of the infinite two-dimensional grid. That is, the nodes of the graph are pairs of integers, and $[(x, y), (x', y')]$ is an edge if and only if $|x - x'| + |y - y'| = 1$. Show that the Hamilton cycle problem for grid graphs is **NP**-complete. (Start with the construction in (b) and embed the graph in the grid. Simulate nodes with small squares, and edges with "tentacles," long strips of width two.)

(d) Conclude from (c) that the *Euclidean* special case of the TSP (the distances are actual Euclidean distances between cities on the plane) is **NP**-complete. (There is a slight problem with defining the Euclidean TSP: To what precision do we need to calculate square roots?)

Part (c) is from

- A. Itai, C. H. Papadimitriou, J. L. Szwarcfiter "Hamilton paths in grid graphs," *SIAM J. Comp., 11*, 3, pp. 676–686, 1982.

Part (d) was first proved in

- C. H. Papadimitriou "The Euclidean traveling salesman problem is **NP**-complete," *Theor. Comp. Sci 4*, pp. 237–244, 1977, and independently in
- M. R. Garey, R. L. Graham, and D. S. Johnson "Some NP-complete geometric problems," in *Proc. 8th Annual ACM Symp. on the Theory of Computing*, pp. 10–22, 1976.

Also, Theorems 9.2 and 9.5 on MAX2SAT and MAX CUT are from

- M. R. Garey, D. S. Johnson, and L. J. Stockmeyer "Some simplified NP-complete graph problems," *Theor. Comp. Sci., 1*, pp. 237-267, 1976.

It is open whether HAMILTON CYCLE is **NP**-complete for *solid* grid graphs, that is, grid graphs without "holes."

9.5.16 In many applications of graph algorithms, such as in vehicle routing and integrated circuit design, the underlying graph is always *planar*, that is, it can be drawn on the plane with no edge crossings. It is of interest to determine which **NP**-complete graph problems retain their complexity when restricted to planar graphs. In this regard, a special case of SAT is especially interesting: The *occurrence graph* of an instance of SAT is the graph that has as nodes all variables and clauses, and has an edge from a variable to a clause if the variable (or its negation) appears in the clause. We say that an instance of SAT is *planar* if its occurrence graph is planar. It was shown in

- D. Lichtenstein "Planar formulae and their uses," *SIAM J. Comp., 11*, pp. 329–393, 1982.

that this special case, called PLANAR SAT, is **NP**-complete even if all clauses have at most three literals, and each variable appears at most five times.

Problem: Use this result to show that INDEPENDENT SET, NODE COVER, and HAMILTON PATH remain **NP**-complete even if the underlying graphs are planar.

How about CLIQUE?

It is open whether BISECTION WIDTH (with unit weights on the edges) is **NP**-complete for planar graphs.

9.5.17 Problem: The *line graph* of a graph $G = (V, E)$ is a graph $L(G) = (E, H)$, where $[e, e'] \in H$ if and only if e and e' are adjacent. Show that HAMILTON PATH is solvable in polynomial time in such graphs.

9.5.18 Problem: The CYCLE COVER problem is this: Given a directed graph, is there a set of node-disjoint cycles that covers all nodes?

(a) Show that the CYCLE COVER problem can be solved in polynomial time (it is a disguise of MATCHING).

(b) Suppose now that we do not allow cycles of length two in our cycle cover. Show that the problem now becomes **NP**-complete. (Reduction from 3SAT. For a similar proof see Theorem 18.3.)

(c) Show that there is an integer k such that the following problem is **NP**-complete: Given an undirected graph, find a cycle cover without cycles of length k or less. (Modify the proof that HAMILTON CYCLE is **NP**-complete.) What is the smallest value of k for which you can prove **NP**-completeness?

(d) Show that the directed Hamilton path problem is polynomial when the directed graph is *acyclic*. Extend to the case in which there are no cycles with fewer than $\frac{n}{2}$ nodes.

9.5.19 Problem: We are given a directed graph with weights w on the edges, two nodes s and t, and an integer B. We wish to find two node-disjoint paths from s to t, both of length at most B. Show that the problem is **NP**-complete. (Reduction from 3SAT. The problem is polynomial when we wish to minimize the sum of the lengths of the two disjoint paths.)

9.5.20 Problem: The CROSSWORD PUZZLE problem is this: We are given an integer n, a subset $B \subseteq \{1, \ldots, n\}^2$ of *black squares*, and a finite *dictionary* $D \subseteq \Sigma^*$. We are asked whether there is a mapping F from $\{1, \ldots, n\}^2 - B$ to Σ such that all *maximal* words of the form $(F(i, j), F(i, j+1), \ldots, F(i, j+k))$ and $(F(i, j), F(i+1, j), \ldots, F(i+k, j))$, are in D. Show that CROSSWORD PUZZLE is **NP**-complete. (It remains **NP**-complete even if $B = \emptyset$.)

9.5.21 Problem: The ZIGSAW PUZZLE problem is the following: We are given a set of polygonal pieces (say, in terms of the sequence of the integer coordinates of the vertices of each). We are asked if there is a way to arrange these pieces on the plane so that (a) no two of them overlap, and (b) their union is a square. Show that ZIGSAW PUZZLE is **NP**-complete. (Start from a version of the TILING problem, Problem 20.2.10, in which some tiles may remain unused. Then simulate tiles by square pieces, with small *square indentations* on each side. The position of the indentations reflects the sort of the tile and forces horizontal and vertical compatibility. Add appropriate small pieces, enough to fill these indentations.)

9.5.22 Problem: Let G be a directed graph. The *transitive closure* of G, denoted

G^*, is the graph with the same set of nodes as G, but which has an edge (u, v) whenever there is a path from u to v in G.

(a) Show that the transitive closure of a graph can be computed in polynomial time. Can you compute it in $\mathcal{O}(n^3)$ time? (Recall Example 8.2. Even faster algorithms are possible.)

(b) The *transitive reduction* of G, $R(G)$, is the graph H with the fewest edges such that $H^* = G^*$. Show that computing the transitive reduction of G is *equivalent* to computing the transitive closure, and hence can be done in $\mathcal{O}(n^3)$ time. (This is from

- A. V. Aho, M. R. Garey, and J. D. Ullman "The transitive reduction of a directed graph," *SIAM J. Comp., 1*, pp. 131–137, 1972.)

(c) Define now the *strong transitive reduction* of G to be the *subgraph* H of G with the fewest edges such that $H^* = G^*$. Show that telling whether the strong transitive reduction of G has K or fewer edges is **NP**-complete. (What is the strong transitive reduction of a strongly connected graph?)

9.5.23 Problem: (a) We are given two graphs G and H, and we are asked whether there is a subgraph of G isomorphic to H. Show that this problem is **NP**-complete.

(b) Show that the problem remains **NP**-complete even if the graph H is restricted to be a tree with the same number of nodes as G.

(c) Show that the problem remains **NP**-complete even if the tree is restricted to have diameter six. (Show first that it is **NP**-complete to tell if a graph with $3m$ nodes has a subgraph which consists of m node-disjoint paths of length two.)

Part (c) and several generalizations are from

- C. H. Papadimitriou and M. Yannakakis "The complexity of restricted spanning tree problems," *J.ACM, 29*, 2, pp. 285-309, 1982.

9.5.24 Problem: We are given a graph G and we are asked if it has as a subgraph a tree T with as many nodes as G (a spanning tree of G, that is), such that the set of *leaves* of T (nodes of degree one) is

(a) of cardinality equal to a given number K.
(b) of cardinality less than a given number K.
(c) of cardinality greater than a given number K.
(d) equal to a given set of nodes L.
(e) a subset of a given set of nodes L.
(f) a superset of a given set of nodes L.

Consider also the variants in which all nodes of T are required to have degree

(g) at most two.
(h) at most a given number K.
(j) equal to an odd number.

Which of these versions are **NP**-complete, and which are polynomial?

9.5.25 Problem: (a) Suppose that in a bipartite graph each set B of boys is adjacent to a set $g(B)$ of girls with $|g(B)| \geq |B|$. Show that the bipartite graph has a matching.

(Start from the Max-Flow Min-Cut Theorem and exploit the relationship between the two problems.)

(b) Show that the INDEPENDENT SET problem for bipartite graphs can be solved in polynomial time.

9.5.26 Interval graphs. Let $G = (V, E)$ be an undirected graph. We say that G is an *interval graph* if there is a path P (a graph with nodes $1, \ldots, m$ for some $m > 1$, and edges of the form $[i, i+1], i = 1, \ldots, m-1$) and a mapping f from V to the set of subpaths (connected subgraphs) of P such that $[v, u] \in E$ if and only if $f(u)$ and $f(v)$ have a node in common.

(a) Show that all trees are interval graphs.

(b) Show that the following problems can be solved in polynomial time for interval graphs: CLIQUE, COLORING, and INDEPENDENT SET.

9.5.27 Chordal graphs. Let $G = (V, E)$ be an undirected graph. We say that G is *chordal* if each cycle $[v_1, v_2, \ldots, v_k, v_1]$ with $k > 3$ distinct nodes has a *chord*, that is, an edge $[v_i, v_j]$ with $j \neq i \pm 1 \bmod k$.

(a) Show that interval graphs (recall the previous problem) are chordal.

A *perfect elimination sequence* of a graph G is a permutation $(v_1, v_2, \ldots, v_n)$ of V such that for all $i \leq n$ the following is true: If $[v_i, v_j], [v_i, v_{j'}] \in E$ and $j, j' > i$, then $[v_j, v_{j'}] \in E$. That is, there is a way of deleting all nodes of the graph, one node at a time, so that the neighborhood of each deleted node is a clique in the remaining graph.

(b) Let A be a symmetric matrix. The *sparsity graph* of A, $G(A)$, has the rows of A as nodes, and an edge from i to j if and only if A_{ij} is non-zero. We say that A has a *fill-in-free elimination* if there is a permutation of rows and columns of A such that the resulting matrix can be made upper diagonal using Gaussian elimination in such a way that *all zero entries of A remain zero*, regardless of the precise values of the nonzero elements.

Finally, we say that G has a *tree model* if there is a tree T and a mapping f from V to the set of subtrees (connected subgraphs) of T such that $[v, u] \in E$ if and only if $f(u)$ and $f(v)$ have a node in common.

(c) Show that the following are equivalent for a graph G:

(i) G is chordal.

(ii) G has a perfect elimination sequence.

(iii) G has a tree model.

(Notice that the equivalence of (i) and (iii) provides a proof of part (a) above. To show that (ii) implies (i), consider a chordless cycle and the first node in it to be deleted. To show that (iii) implies (ii), consider all nodes u of G such that $f(u)$ contains a leaf of T. That (i) implies (iii) is tedious.)

(d) Show that the following problems can be solved in polynomial time for chordal graphs: CLIQUE, COLORING, and INDEPENDENT SET.

(e) The clique number $\omega(G)$ of a graph G is the number of nodes in its largest clique.

The chromatic number $\chi(G)$ is the minimum number of colors that are required to color the nodes of the graph so that no two adjacent nodes have the same color. Show that for all G $\chi(G) \geq \omega(G)$.

A graph G is called *perfect* if for all of its subgraphs G' $\chi(G') = \omega(G')$.

(f) Show that interval graphs, chordal graphs, and bipartite graphs are perfect.

For much more on perfect graphs, their special cases, and their positive algorithmic properties, see

- M. C. Golumbic *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, 1980.

9.5.28 Problem: Show that 3-COLORING remains **NP**-complete even when the graph is planar. (Again, we must replace crossings with suitable graphs. This **NP**-completeness result is remarkable because telling whether any graph can be colored with two colors is easy (why?), and coloring a planar graph with four colors is always possible, see

- K. Appel and W. Haken "Every planar map is 4-colorable," *Illinois J. of Math., 21*, pp. 429–490 and 491–567, 1977.)

9.5.29 Disjoint paths in graphs. In the problem DISJOINT PATHS we are given a directed graph G, and a set of pairs of nodes $\{(s_1, t_1), \ldots, (s_k, t_k)\}$. We are asked whether there are *node-disjoint paths* from s_i to t_i, $i = 1, \ldots, k$.

(a) Show that DISJOINT PATHS is **NP**-complete. (From 3SAT. The graph you construct has for each variable two endpoints and two separate parallel paths between them, and for each clause two endpoints connected by three paths that correspond to the literals in the clause. These two kinds of paths should intersect in the appropriate way.)

(b) Show that DISJOINT PATHS is **NP**-complete even if the graph is planar. (You may wish to start from PLANAR SAT (see 9.5.16 above). Or, introduce new endpoints at each crossing.)

(c) Show that DISJOINT PATHS is **NP**-complete even if the graph is undirected. These results were first shown in

- J. F. Lynch "The equivalence of theorem proving and the interconnection problem," *ACM SIGDA Newsletter, 5*, 3, pp. 31–36, 1975.

(d) Show that the special case of the DISJOINT PATHS problem, in which all sources coincide ($s_1 = s_2 = \ldots = s_k$), is in **P**.

Let $k > 1$, and define k DISJOINT PATHS to be the special case of the problem in which there are exactly k pairs.

(e) Show that the 2 DISJOINT PATHS problem is **NP**-complete. (This is a clever **NP**-completeness proof due to

- S. Fortune, J. E. Hopcroft, and J. Wyllie "The directed subgraph homeomorphism problem," *Theor. Comp. Sci., 10*, pp. 111–121, 1980.)

(f) Let H be a directed graph with k edges. The DISJOINT H-PATHS problem is the special case of k DISJOINT PATHS, in which the s_i's and t_i's are required to coincide

as the heads and tails of the edges of H. For example, if H is the directed graph with two nodes $1, 2$ and two edges $(1, 2), (2, 1)$, the DISJOINT H-PATHS problem is this: Given a directed graph and two nodes a and b, is there a simple cycle (with no repetitions of nodes) involving a and b? Using the results in (d) and (e) above show that the DISJOINT H-PATHS problem is always **NP**-complete, unless H is a tree of depth one (see part (d) above), in which case it is in **P**.

The undirected special case of k DISJOINT PATHS is in **P** *for all* k. See

- N. Robertson and P. D. Seymour "Graph minors XIII: The disjoint paths problem," thirteenth part of a series of twenty papers, of which nineteen appear in *J. Combinatorial Theory, Ser. B, 35*, 1983, and thereafter; the second paper in the series appeared in the *J. of Algorithms, 7*, 1986.

This powerful and surprising result is just one step along the way of proving perhaps the most important and sweeping result in algorithmic graph theory to-date, namely that *all graph-theoretic properties that are closed under minors are in* **P**. We say that a graph-theoretic property is closed under minors if, whenever G has the property, then so does any graph that results from G by (a) deleting a node, and (b) collapsing two adjacent nodes. See the sequence of papers referenced above.

(g) Suppose that H is the directed graph consisting of a single node and *two self-loops*. Then the DISJOINT H-PATHS problem is this: Given a directed graph and a node a, are there two disjoint cycles through a? By (f) above, it is **NP**-complete. Show that it can be solved in polynomial time for *planar* directed graphs.

In fact, it turns out that the DISJOINT H-PATHS problem can be solved in polynomial time for planar graphs *for any* H; see

- A. Schrijver "Finding k disjoint paths in a directed planar graph," Centrum voor Wiskunde en Informatica Report BS-R9206, 1992.

9.5.30 The BANDWIDTH MINIMIZATION problem is this: We are given an undirected graph $G = (V, E)$ and an integer B. We are asked whether there is a permutation $(v_1, v_2, \ldots, v_n)$ of V such that $[v_i, v_j] \in E$ implies $|i - j| < B$. This problem is **NP**-complete, see

- C. H. Papadimitriou "The NP-completeness of the bandwidth minimization problem," *Computing, 16*, pp. 263–270, 1976, and
- M. R. Garey, R. L. Graham, D. S. Johnson, and D. E. Knuth "Complexity results for bandwidth minimization," *SIAM J. Appl. Math., 34*, pp. 477–495, 1978.

In fact, the latter paper shows that BANDWIDTH MINIMIZATION remains **NP**-complete *even if* G *is a tree.*

Problem: (a) Show that *all other graph-theoretic problems* we have seen in this chapter, including this section, can be solved in polynomial time if the given graph is a tree.

(b) Show that the BANDWIDTH MINIMIZATION problem with B fixed to any bounded integer can be solved in polynomial time. (This follows of course from the result on minor-closed properties discussed in 9.5.29 above; but a simple dynamic programming algorithm is also possible.)

9.5.31 Pseudopolynomial algorithms and strong NP-completeness. There is a difficulty in formalizing the concept of pseudopolynomial algorithms and strong **NP**-completeness discussed in Section 9.4. The difficulty is this: Strictly speaking, inputs are dry, uninterpreted strings operated on by Turing machines. The fact that certain parts of the input encode binary integers should be completely transparent to our treatment of algorithms and their complexity. How can we speak about the size of integers in the input without harming our convenient abstraction and representation-independence?

Suppose that each string $x \in \Sigma^*$ has, except for its length $|x|$, another integer value $\text{NUM}(x)$ associated with it. All we know about this integer is that, for all x, $\text{NUM}(x)$ can be computed in polynomial time from x, and that $|x| \leq \text{NUM}(x) \leq 2^{|x|}$. We say that a Turing machine operates in *pseudopolynomial time* if for all inputs x the number of steps of the machine is $p(\text{NUM}(x))$ for some polynomial $p(n)$. We say that a language L is *strongly* **NP**-*complete* if there is a polynomial $q(n)$ such that the following language is **NP**-complete: $L_{q(n)} = \{x \in L : \text{NUM}(x) \leq q(|x|)\}$.

Problem: (a) Find a plausible definition of $\text{NUM}(x)$ for KNAPSACK. Show that the algorithm in Proposition 9.4 is pseudopolynomial.

(b) Find a plausible definition of $\text{NUM}(x)$ for BIN PACKING. Show that BIN PACKING is strongly **NP**-complete.

(c) Show that, if there is a pseudopolynomial algorithm for a strongly **NP**-complete problem, then $\mathbf{P} = \mathbf{NP}$.

Naturally, the pseudopolynomiality vs. strong **NP**-completeness information we obtain is only as good as our definition of $\text{NUM}(x)$; and in any problem one can come up with completely implausible and far-fetched definitions of $\text{NUM}(x)$. The point is, *we can always come up with a good one* (namely, "the largest integer encoded in the input"), for which the dichotomy of Part (c) above is both meaningful and informative.

9.5.32 Problem: Let $k \geq 2$ be a fixed integer. The k-PARTITION problem is the following special case of BIN PACKING (recall Theorem 9.11): We are given $n = km$ integers $a_1, \ldots, a_n$, adding up to mC, and such that $\frac{C}{k+1} < a_i < \frac{C}{k-1}$ for all i. That is, the numbers are such that their sum fits exactly in m bins, but no $k+1$ of them fit into one bin, neither can any $k-1$ of them fill a bin. We are asked whether we can find a partition of these numbers into m groups of k, such that the sum in each group is precisely C.

(a) Show that 2-PARTITION is in **P**.

(b) Show that 4-PARTITION is **NP**-complete. (The proof of Theorem 9.11 basically establishes this.)

(c) Show that 3-PARTITION is **NP**-complete. (This requires a clever reduction of 4-PARTITION to 3-PARTITION.)

Part (c), the **NP**-completeness of 3-PARTITION, is from

- M. R. Garey and D. S. Johnson "Complexity results for multiprocessor scheduling with resource constraints," *SIAM J. Comp.*, 4, pp. 397–411, 1975.

9.5.33 (a) Show that the following problem is **NP**-complete: Given n integers adding

up to $2K$, is there a subset that adds up to exactly K? This is known as the PARTITION problem, not to be confused with the k-PARTITION problem above. (Start from KNAPSACK, and add appropriate new items.)

(b) INTEGER KNAPSACK is this problem: We are given n integers and a goal K. We are asked whether we can choose *zero, one, or more* copies of each number so that the resulting multiset of integers adds up to K. Show that this problem is **NP**-complete. (Modify an instance of the ordinary KNAPSACK problem so that each item can be used at most once.)

9.5.34 Linear and integer programming. INTEGER PROGRAMMING is the problem of deciding whether a given system of linear equations has a nonnegative integer solution. It is of course **NP**-complete, as just about *all* **NP**-complete problems easily reduce to it... Actually, the difficult part here is showing that it is in **NP**; but it can be done, see

- C. H. Papadimitriou "On the complexity of integer programming", *J.ACM, 28*, 2, pp. 765-769, 1981.

In fact, in the paper above it is also shown that there is a pseudopolynomial algorithm for INTEGER PROGRAMMING when the number of equations is bounded by a constant, thus generalizing Proposition 9.4. (Naturally, the general INTEGER PROGRAMMING problem is strongly **NP**-complete.)

A different but equivalent formulation of INTEGER PROGRAMMING is in terms of a system of *inequalities* instead of equalities, and variables unrestricted in sign. For this form we have a more dramatic result: When the number of variables is bounded by a constant, there is a *polynomial-time* algorithm for the problem, based on the important *basis reduction technique*; see

- A. K. Lenstra, H. W. Lenstra, and L. Lovász "Factoring polynomials with rational coefficients," *Math. Ann, 261*, pp. 515–534, 1982, and
- M. Grötschel, L. Lovász, and A. Schrijver *Geometric Algorithms and Combinatorial Optimization*, Springer, Berlin, 1988.

In contrast, linear programming (the version in which we are allowed to have fractional solutions), is much easier: Despite the fact that the classical, empirically successful, and influential *simplex method*, see

- G. B. Dantzig *Linear Programming and Extensions*, Princeton Univ. Press, Princeton, N.J., 1963,

is exponential in the worst-case, polynomial-time algorithms have been discovered. The first polynomial algorithm for linear programming was the *ellipsoid method*

- L. G. Khachiyan "A polynomial algorithm for linear programming," *Dokl. Akad. Nauk SSSR, 244*, pp. 1093–1096, 1979. English Translation *Soviet Math. Doklad 20*, pp. 191–194, 1979;

while a more recent algorithm seems to be much more promising in practice:

- N. Karmarkar "A new polynomial-time algorithm for linear programming," *Combinatorica, 4*, pp. 373 395, 1984.

See also the books

- A. Schrijver *Theory of Linear and Integer Programming*, Wiley, New York, 1986, and
- C. H. Papadimitriou and K. Steiglitz *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, New Jersey, 1982.

Problem: (a) Show that any instance of SAT can be expressed very easily as an instance of INTEGER PROGRAMMING with inequalities. Conclude that INTEGER PROGRAMMING is **NP**-complete even if the inequalities are known to have a fractional solution. (Start with an instance of SAT with at least two distinct literals per clause.)

(b) Express the existence of an integer flow of value K in a network with integer capacities as a set of linear inequalities.

(c) Is the MAX FLOW problem a special case of linear, or of integer programming? (On the surface it appears to be a special case of integer programming, since integer flows are required; but a little thought shows that the optimum solution will always be integer anyway—assuming all capacities are. So, the integrality constraint is superfluous.)

CHAPTER 10

coNP AND FUNCTION PROBLEMS

The asymmetry of nondeterminism opens up intriguing possibilities for the classification of problems, including some of the most classical problems in mathematics.

10.1 NP AND coNP

If **NP** is the class of problems that have succinct certificates (recall Theorem 9.1), then **coNP** must contain those problems that have *succinct disqualifications*. That is, a "no" instance of a problem in **coNP** possesses a short proof of its being a "no" instance; and only "no" instances have such proofs.

Example 10.1: VALIDITY of Boolean expressions is a typical problem in **coNP**. We are given a Boolean expression ϕ, and we are asked whether it is valid, satisfiable by all truth assignments. If ϕ is *not* a valid formula, then it can be disqualified very succinctly: By exhibiting a truth assignment that does not satisfy it. No valid formula has such a disqualification.

For another example, HAMILTON PATH COMPLEMENT is the set of all graphs that have no Hamilton path. Since it is the complement of the HAMILTON PATH problem[†], it is in **coNP**. Here the disqualification is, naturally enough, a Hamilton path: All "no" instances of HAMILTON PATH COMPLEMENT, and only these, have one. We should also mention here SAT COMPLE-

[†] Recall that we say two languages are complements of each other if they are disjoint and their union is, not necessarily the set of all strings, but some other, trivial to recognize set; in this case, the set of all strings that are legitimate encodings of graphs.

MENT; but in some sense we already have: VALIDITY is just SAT COMPLEMENT *with the expression negated* (recall Figure 4.1).

Needless to say, all problems that are in **P** are *ex officio* in **coNP**, since $\mathbf{P} \subseteq \mathbf{NP}$ is a deterministic complexity class, and therefore closed under complement. □

VALIDITY and HAMILTON PATH COMPLEMENT are examples of **coNP**-*complete problems*. We claim that any language L in **coNP** is reducible to VALIDITY. In proof, if $L \in \mathbf{coNP}$ then $\overline{L} \in \mathbf{NP}$, and thus there is a reduction R from $\overline{L}$ to SAT. For any string x we have $x \in \overline{L}$ if and only if $R(x)$ is satisfiable. The reduction from L to VALIDITY is this: $R'(x) = \neg R(x)$. The proof HAMILTON PATH COMPLEMENT is very similar. More generally, we have:

Proposition 10.1: If L is **NP**-complete, then its complement $\overline{L} = \Sigma^* - L$ is **coNP**-complete. □

Whether $\mathbf{NP} = \mathbf{coNP}$ is another fundamental question that is as important as it is unresolved (you will encounter many more of these). Of course, if after all it turns out that $\mathbf{P} = \mathbf{NP}$, then $\mathbf{NP} = \mathbf{coNP}$ will also hold, since **P** is closed under complement. Conceivably it could be that $\mathbf{P} \neq \mathbf{NP}$, and still $\mathbf{NP} = \mathbf{coNP}$ (in Chapter 14 we shall even introduce a formal notion of "conceivably" to deal with such speculations). But it is strongly believed that **NP** and **coNP** are different. All efforts to find systematic ways for deriving short proofs of VALIDITY, for example, have failed (and there is a long history of such efforts, for example along the lines of resolution, recall Problem 4.4.10; see 10.10.4.4). Similarly, we know of no characterizations of non-Hamiltonian graphs that would allow us to demonstrate the absence of a Hamilton path succinctly.

The problems in **coNP** that are **coNP**-complete are the least likely to be in **P**. What is more, *they are also the least likely to be in* **NP**:

Proposition 10.2: If a **coNP**-complete problem is in **NP**, then $\mathbf{NP} = \mathbf{coNP}$.

Proof: We shall show that if $L \in \mathbf{NP}$ is **coNP**-complete then $\mathbf{coNP} \subseteq \mathbf{NP}$—the other direction follows then easily from symmetry. Consider a language $L' \in \mathbf{coNP}$. Since L is **coNP**-complete, there is a reduction R from L' to L. Hence, a polynomial-time nondeterministic machine for L' on input x would first compute $R(x)$, and supply it to the nondeterministic machine for L. □

The similarities and symmetry between **NP** and **coNP** never stop: There is a logical characterization of **coNP** as the set of graph-theoretic properties that can be expressed in *universal* second-order logic (Problem 10.4.3; recall Fagin's theorem 8.3).

Example 10.2: A most important member of **coNP** is the problem PRIMES, asking whether an integer N given in binary is a prime number. The string that disqualifies a prime is, of course, a *divisor* other than itself and one. Obviously,

all numbers that are composites (that is, not primes), and only these, have such a disqualification; it is clearly a succinct one.

The obvious $\mathcal{O}(\sqrt{N})$ algorithm for solving PRIMES (see Problem 10.4.8) does not fool anyone: It is *pseudopolynomial*, not polynomial, because its time bound is not polynomial in the length of the input, which is $\log N$. At present, it is not known whether there is a polynomial algorithm for PRIMES; in this respect PRIMES resembles the **coNP**-complete problems such as VALIDITY. However, the similarity stops here. For PRIMES we have a number of positive complexity results (see Chapter 11 and the references there) which provide a sharp contrast between this problem and the **NP**-complete and **coNP**-complete problems. For example, we shall see later in this chapter that problem PRIMES, besides being in **coNP**, *is also in* **NP**. And in the next chapter we shall show that there is a fast *randomized* algorithm for PRIMES. Finally, in 10.4.11 we mention further positive algorithmic results for this important problem. □

The Class NP ∩ coNP

So, problems in **NP** have succinct certificates, whereas problems in **coNP** have succinct disqualifications. **NP** ∩ **coNP** is the class of problems that have both! That is, problems in **NP** ∩ **coNP** have the following property: Each instance either has a succinct certificate (in which case it is a "yes" instance) or it has a succinct disqualification (in which case it is a "no" instance). No instance has both. The nature of certificates and disqualifications will be very different, and different algorithms will be used to validate them. But exactly one of the two always exists. Obviously, any problem in **P** is also in **NP** ∩ **coNP**, but there are examples of problems in **NP** ∩ **coNP** that are not known to be in **P** (in contrast, recall that, at the level of decidability, **RE** ∩ **coRE** = **R**, Proposition 3.4). For example, we shall soon see that PRIMES is in **NP** ∩ **coNP**. But before showing this result, which requires the introduction of certain number-theoretic concepts, we shall first look at a general way of showing that *optimization* problems of a certain kind are in **NP** ∩ **coNP**.

Recall the MAX FLOW problem in Chapter 1. We know that this problem is in **P**, but let us for a moment ignore this, and recall another observation (Problem 1.4.11): The value of the maximum flow in a network is the same as the value of the *minimum cut*, that is, the smallest sum of capacities over all sets of edges that disconnect t from s (see Figure 1.2). Now, both of these problems are optimization problems, and they can be turned into decision problems as follows: MAX FLOW (D) asks, given a network and a goal K, whether there is a flow from s to d of value K. MIN CUT (D) asks, given a network and a budget B, whether there is a set of edges of total capacity B or less, such that deleting these edges disconnects d from s. The *max flow-min cut theorem* (Problem 1.4.11) states that a network has a flow of value K if and only if it does not have a

cut of capacity $K + 1$. In other words, each of the two problems MAX FLOW (D) and MIN CUT (D) is in **NP**, and each is easily reducible to the *complement* of the other. When this happens, we say that the two problems (typically the decision versions of a maximization and of a minimization problem) are *dual* to each other. See Problem 10.4.5 for more on duality. Duality immediately implies that both problems are in **NP** $\cap$ **coNP**.

10.2 PRIMALITY

A number p is prime if $p > 1$ and all other numbers (except for 1) fail to divide it. This definition, with its obvious "for all" quantifier, immediately suggests that the problem PRIMES is in **coNP** (recall the discussion in Example 10.2). In this section we shall show that PRIMES is in **NP**. To do this, we must develop an alternative characterization of primes, one that has only "exists" quantifiers. Here it is:

Theorem 10.1: A number $p > 1$ is prime if and only if there is a number $1 < r < p$ such that $r^{p-1} = 1 \bmod p$, and furthermore $r^{\frac{p-1}{q}} \neq 1 \bmod p$ for all prime divisors q of $p - 1$.

In the next subsection we shall develop enough number theory to prove Theorem 10.1. But for now, let us notice that indeed it is the "alternative characterization of primes" that we seek:

Corollary (Pratt's Theorem): PRIMES is in **NP** $\cap$ **coNP**.

Proof: We know that PRIMES is in **coNP**. To show that it is in **NP**, we shall show that any prime possesses a certificate of a kind that is missing from all composites, and one that is polynomially succinct and polynomially checkable.

Suppose that p is a prime. Clearly, p's certificate will contain r in the theorem above. We claim that it is easy to check whether $r^{p-1} = 1 \bmod p$ in time polynomial in *the logarithm* of p, $\ell = \lceil \log p \rceil$. To see this, first notice that multiplication modulo p can be carried out by an ordinary multiplication, followed by an ordinary division by p. And the "grammar school" methods for multiplying and dividing ℓ-bit integers take $\mathcal{O}(\ell^2)$ elementary steps for ℓ-bit integers (Problem 10.4.7). In order to calculate $r^{p-1} \bmod p$ we do not perform the $p-2$ multiplications suggested (this would *not* be polynomial in ℓ): Instead we square r repeatedly ℓ times, thus obtaining $r^2, r^4, \ldots, r^{2^\ell}$, always modulo p (otherwise, the numbers would grow out of control). Then with at most ℓ more multiplications modulo p (as suggested by the binary expansion of $p - 1$) we obtain $r^{p-1} \bmod p$, and check that it is equal to 1. The time required is $\mathcal{O}(\ell^3)$, a polynomial.

But unfortunately r itself is not a sufficient certificate of primality: Notice that $20^{21-1} = 1 \bmod 21$ (just observe that 20 is really -1 modulo 21, and thus its square is 1; raising 1 to the tenth power results in 1), and still 21 is no prime. We must also provide, as part of the certificate of p, *all prime divisors of* $p-1$ (recall the statement of Theorem 10.1). So, our proposed certificate of p's primality is $C(p) = (r; q_1, \ldots, q_k)$, where the q_i's are all the prime divisors of $p-1$. Once they are given, we can first check that $r^{p-1} = 1 \bmod p$, and that $p-1$ indeed is reduced to one by repeated divisions by the q_i's. Then, we check that indeed $r^{\frac{p-1}{q_i}} \neq 1 \bmod p$ for each i, by the exponentiation algorithm in the previous paragraph. All this can be done in polynomial time.

This certificate is still incomplete. Here is a "certificate" as above that 91 is a prime: $C(91) = (10; 2, 45)$. Indeed, $10^{90} = 1 \bmod 91$ (to see this, hust notice that $10^6 = 1 \bmod 91$), and repeated division of 90 by 2 and 45 indeed reduces it to one. Finally, $10^{\frac{90}{2}} = 90 \neq 1 \bmod 91$, and $10^{\frac{90}{45}} = 9 \neq 1 \bmod 91$. So, the "certificate" checks. Still, $91 = 7 \times 13$ is not a prime! The "certificate" $(10; 2, 45)$ is misleading, *because* 45 *is not a prime!*

The problem is that $C(p) = (r; q_1, \ldots, q_k)$ does not convince us that the q_i's are indeed primes. The solution is simple: For each q_i in $C(p)$ also supply *its primality certificate*, which must exist by induction. The recursion stops when $p = 2$, and there are no prime divisors of $p-1$. That is, the certificate that p is a prime is the following: $C(p) = (r; q_1, C(q_1), \ldots, q_k, C(q_k))$. For example, here is $C(67)$: $(2; 2, (1), 3, (2; 2, (1)), 11, (8; 2, (1), 5, (3; 2, (1))))$.

If p is a prime, then by the theorem and the induction an appropriate certificate $C(p)$ exists. And if p is not a prime, then Theorem 10.1 implies that no legitimate certificate exists.

Furthermore, $C(p)$ is succinct: We claim that its total length is at most $4 \log^2 p$. The proof is by induction on p. It certainly holds when $p = 2$ or $p = 3$. For a general p, $p-1$ will have $k < \log p$ prime divisors $q_1 = 2, \ldots, q_k$. The certificate $C(p)$ will thus contain, besides the two parentheses and $2k < 2 \log p$ more separators, the number r (at most $\log p$ bits), 2 and its certificate (1) (five bits) the q_i's (also at most $2 \log p$ bits), and the $C(q_i)$'s. Now, by induction $|C(q_i)| \leq 4 \log^2 q_i$, and thus we have

$$|C(p)| \leq 4 \log p + 5 + 4 \sum_{i=2}^{k} \log^2 q_i. \tag{1}$$

The logarithms of the $k-1$ q_i's add up to $\log \frac{p-1}{2} < \log p - 1$, and thus the sum of squares in (1) is at most $(\log p - 1)^2$. Substituting, we get $|C(p)| \leq 4 \log^2 p + 9 - 4 \log p$, which for $p \geq 5$ is less than $4 \log^2 p$. The induction is complete.

Finally, $C(p)$ can be checked in polynomial time. The algorithm that checks $C(p)$ requires time $\mathcal{O}(n^3)$ for calculating the $r^{p-1} \bmod p$ and $r^{\frac{p-1}{q_i}} \bmod p$, where n is the number of bits of p, plus the time required for verifying that the provided primes reduce $p-1$ to one, and for checking the embedded certificates. A calculation completely analogous to that of the previous paragraph shows that the total time is $\mathcal{O}(n^4)$. □

Primitive Roots Modulo a Prime

In this subsection we shall prove Theorem 10.1. Let us recall some basic definitions and notations concerning numbers and divisibility. We shall only consider positive integers. The symbol p will always denote a prime. We say that m divides n if $n = mk$ for some integer k; we write $m|n$, and m is called a divisor of n. Every number is the product of primes (not necessarily distinct, of course). The greatest common divisor of m and n is denoted (m,n); if $(m,n) = 1$ then m and n are called *prime* to each other. If n is any number, we call the numbers $0, 1, \ldots, n-1$ the *residues* modulo n. All arithmetic operations in this subsection will be reduced modulo some number, so that we are always dealing with residues modulo that number.

The number r whose existence is postulated in Theorem 10.1 is called a *primitive root* of p, and all primes have several of those. However, the primitive roots of p are well-hidden among the residues of p, so we shall examine these numbers first. Let $\Phi(n) = \{m : 1 \leq m < n, (m,n) = 1\}$ be the set of all numbers less than n that are prime to n. For example, $\Phi(12) = \{1,5,7,11\}$ and $\Phi(11) = \{1,2,3,\ldots,10\}$. We define the *Euler's function* of n to be $\phi(n) = |\Phi(n)|$. Thus, $\phi(12) = 4$, and $\phi(11) = 10$. We adopt the convention that $\phi(1) = 1$. Notice that $\phi(p) = p-1$ (remember, p always denotes a prime).

More generally, one can compute the Euler function as follows: Consider the n numbers $0, 1, \ldots, n-1$. They are all candidate members of $\Phi(n)$. Suppose that $p|n$. Then p "cancels" one in every p candidates, leaving $n(1-\frac{1}{p})$ candidates. If now q is another prime divisor of n, then it is easy to see that it cancels one every q of the remaining candidates leaving $n(1-\frac{1}{p})(1-\frac{1}{q})$ (this way, we do not double-count the cancellations of the $\frac{n}{pq}$ multiples of pq). And so on:

Lemma 10.1: $\phi(n) = n\prod_{p|n}(1-\frac{1}{p})$. □

This is not very useful computationally, but it does expose the "multiplicative" nature of this function:

Corollary 1: If $(m,n) = 1$, then $\phi(mn) = \phi(m)\phi(n)$. □

Another useful corollary of Lemma 10.1 considers the case in which n is

the product of *distinct* primes $p_1, \ldots, p_k$. In this case, $\phi(n) = \prod_{i=1}^{k}(p_i - 1)$. But this means that there is a one-to-one correspondence between k-tuples of residues $(r_1, \ldots, r_k)$, where $r_i \in \Phi(p_i)$ with the residues $r \in \Phi(n)$, via the mapping $r_i = r \bmod p_i$ (the inverse mapping is a little more complicated, see Problem 10.4.9).

Corollary 2 (The Chinese Remainder Theorem): If n is the product of distinct primes $p_1, \ldots, p_k$, for each k-tuples of residues $(r_1, \ldots, r_k)$, where $r_i \in \Phi(p_i)$, there is a unique $r \in \Phi(n)$, where $r_i = r \bmod p_i$. □

The next property of the Euler function is quite remarkable, and so is its proof:

Lemma 10.2: $\sum_{m|n} \phi(m) = n$.

Proof: Let $\prod_{i=1}^{\ell} p_i^{k_i}$ be the prime factorization of n, and consider the following product:

$$\prod_{i=1}^{\ell}(\phi(1) + \phi(p_i) + \phi(p_i^2) + \ldots + \phi(p_i^{k_i})).$$

It is easy to see, calculating ϕ from Lemma 10.1 in the case there is only one prime divisor, that the ith factor of this product is $[1 + (p_i - 1) + (p_i^2 - p_i) + \ldots + (p_i^{k_i} - p_i^{k_1 - 1})]$ which is $p_i^{k_i}$. Thus, this product is just a complicated way to write n.

On the other hand, if we expand the product, we get the sum of many terms, *one for each divisor of* n. The term corresponding to $m = \prod_{i=1}^{\ell} p_i^{k_i'}$, where $0 \leq k_i' \leq k_i$, is $\prod_{i=1}^{\ell} \phi(p_i^{k_i'})$. However, this is the product of the Euler functions of powers of distinct primes, and so the Corollary to Lemma 10.1 applies $\ell - 1$ times to yield $\phi(\prod p_i^{k_i'}) = \phi(m)$. Adding all the terms, we get the result. □

We shall use Lemma 10.2 in order to show that every prime p has many primitive roots modulo p. As a first step, it is easy to see that all elements of $\Phi(p)$ fulfill the first requirement of the definition of a primitive root:

Lemma 10.3 (Fermat's Theorem): For all $0 < a < p$, $a^{p-1} = 1 \bmod p$.

Proof: : Consider the set of residues $a \cdot \Phi(p) = \{a \cdot m \bmod p : m \in \Phi(p)\}$. It is easy to see that this set is the same as $\Phi(p)$, that is, multiplication by a just *permutes* $\Phi(p)$. Because, otherwise we would have $am = am' \bmod p$ for $m > m'$, where $m, m' \in \Phi(p)$, and thus $a(m - m') = 0 \bmod p$. But this is absurd: a and $m - m'$ are integers between 1 and $p - 1$, and still their product is divisible by the prime p. So, $a \cdot \Phi(p) = \Phi(p)$. Multiply now all numbers in both sets: $a^{p-1}(p-1)! = (p-1)! \bmod p$, or $(a^{p-1} - 1)(p-1)! = 0 \bmod p$. Thus, two integers have a product that is divisible by p, and so at least one must be divisible by p. But $(p - 1)!$ cannot be divisible by p, and so it must be that $(a^{p-1} - 1)$ is. We conclude that $a^{p-1} = 1 \bmod p$. □

In fact, we can apply the same argument to any $\Phi(n)$, where n is not necessarily a prime, and obtain the following generalization (it *is* a generalization, because $\phi(p) = p - 1$).

Corollary: For all $a \in \Phi(n)$, $a^{\phi(n)} = 1 \bmod n$. □

Unfortunately, not all elements of $\Phi(p)$ are primitive roots. Consider $\Phi(11) = \{1, 2, \ldots, 10\}$. Of its 10 elements, only four are primitive roots modulo 11. For example, the powers of 3 mod 11 are these, in increasing exponent: $(3, 9, 5, 4, 1, 3, 9, \ldots)$. This disqualifies 3 from being a primitive root, because $3^{\frac{10}{2}} = 3^5 = 1 \bmod 11$; similarly, 10 cannot be a primitive root because $10^{\frac{10}{5}} = 10^2 = 1 \bmod 11$ (here 2 and 5 are the prime divisors of 10, the q's in Theorem 10.1). On the other hand, 2 is a primitive root modulo 11. The powers of 2 mod 11 are these: $(2, 4, 8, 5, 10, 9, 7, 3, 6, 1, 2, 4, \ldots)$. Therefore, 2 is indeed a primitive root, as $2^{\frac{10}{2}}$ and $2^{\frac{10}{5}}$ are not 1 mod 11.

In general, if $m \in \Phi(p)$, the *exponent* of m is the least integer $k > 0$ such that $m^k = 1 \bmod p$. All residues in $\Phi(p)$ have an exponent, because if the powers of $s \in \Phi(p)$ repeat without ever reaching one, we again have that $s^i(s^{j-i} - 1) = 0 \bmod p$, and thus $j - i$ is an exponent. If the exponent of m is k, then the only powers of m that are 1 mod p are the ones that are multiples of k. It follows from Lemma 10.3 that *all exponents divide* $p - 1$. We are now in a position to understand the requirements of Theorem 10.1 a little better: By requiring that $r^{\frac{p-1}{q}} \neq 1 \bmod p$ for all prime divisors q of $p - 1$, we rule out exponents that are *proper* divisors of $p - 1$, and so demand that the exponent of r be $p - 1$ itself.

Let us fix p, and let $R(k)$ denote the total number of residues in $\Phi(p)$ that have exponent k. We know that $R(k) = 0$ if k does not divide $p - 1$. But, if it does, how large can $R(k)$ be? A related question is, how many roots can the equation $x^k = 1$ have modulo p? It turns out that residues behave like the real and complex numbers in this respect:

Lemma 10.4: Any polynomial of degree k that is not identically zero has at most k distinct roots modulo p.

Proof: By induction on k. It is clear when $k = 0$ the result holds, so suppose that it holds for up to degree $k-1$. Suppose that $\pi(x) = a_k x^k + \ldots + a_1 x + a_0$ has $k + 1$ distinct roots modulo p, say $x_1, \ldots, x_{k+1}$. Then consider the polynomial $\pi'(x) = \pi(x) - a_k \prod_{i=1}^{k}(x - x_i)$. Obviously, $\pi'(x)$ is of degree at most $k - 1$, because the coefficients of x^k cancel. Also, $\pi'(x)$ is not identically zero, because $\pi'(x_{k+1}) = -a_k \prod_{i=1}^{k}(x_{k+1} - x_i) \neq 0 \bmod p$, since all x_i's are supposed distinct modulo p. Therefore, by the induction hypothesis $\pi'(x)$ has $k - 1$ or fewer distinct roots. But notice that $x_1, \ldots, x_k$ are all roots of $\pi'(x)$, and they are indeed distinct by hypothesis. □

So, there are at most k residues of exponent k. Suppose there is one, call

it s. Then $(1, s, s^2, \ldots, s^{k-1})$ are all distinct (because $s^i = s^j \bmod p$ with $i < j$ implies $s^i(s^{j-i} - 1) = 0 \bmod p$, and thus s is of exponent $j - i < k$). All these k residues have the property that $(s^i)^k = s^{ki} = 1^i = 1 \bmod p$. So, these are all the possible solutions of $x^k = 1$. Still, not all these numbers have exponent k. For example 1 has exponent one. Also, if $\ell < k$ and $\ell \notin \Phi(k)$ (that is, if ℓ and k have a nontrivial common divisor, call it d), then clearly $(s^\ell)^{\frac{k}{d}} = 1 \bmod p$, and so s^ℓ has exponent $\frac{k}{d}$ or less. Thus, if s^ℓ has exponent k, this means that $\ell \in \Phi(k)$. *We have shown that* $R(k) \leq \phi(k)$.

We are very close to proving the difficult direction of Theorem 10.1: So, $R(k)$ is zero when k does not divide $p - 1$, and at most $\phi(k)$ otherwise. But all $p - 1$ residues have an exponent. So,

$$p - 1 = \sum_{k|p-1} R(k) \leq \sum_{k|p-1} \phi(k) = p - 1,$$

the last equality provided by Lemma 10.2. Hence we must have that $R(k) = \phi(k)$ for all divisors of $p - 1$. In particular, $R(p - 1) = \phi(p - 1) > 0$, and thus p has at least one primitive root.

Conversely, suppose that p is not a prime (but we shall still call it p, violating for a moment our convention). We shall show that there is no r as prescribed by the theorem. Suppose that, indeed, $r^{p-1} = 1 \bmod p$. We know from the Corollary to Lemma 10.3 that $r^{\phi(p)} = 1 \bmod p$, and since p is not a prime $\phi(p) < p - 1$. Consider now the smallest k such that $r^k = 1 \bmod p$. It is clear that k divides both $p - 1$ and $\phi(p)$, and thus it is strictly smaller than $p - 1$. Let q be a prime divisor of $\frac{p-1}{k}$; then $r^{\frac{p-1}{q}} = 1 \bmod p$, violating the condition of the theorem, and thus completing its proof. □

Primitive roots can exist even when the modulo is not a prime. We say that r is a primitive root modulo n if $r^k \neq 1 \bmod n$ for all divisors k of $\phi(n)$. For example, we can show the following (for the proof see Problem 10.4.10):

Proposition 10.3: Every prime square p^2, where $p > 2$, has a primitive root. □

10.3 FUNCTION PROBLEMS

Early on in this book we decided to study computation and complexity mainly in terms of *languages* encoding *decision problems*. Have we lost anything by doing this? Obviously, the computational tasks that need to get solved in practice are not all of the kind that take a "yes" or "no" answer. For example, we may need to find a *satisfying truth assignment* of a Boolean expression, not just to tell whether the expression is satisfiable; in the traveling salesman problem we want *the optimal tour*, not just whether a tour within a given budget exists; and so on. We call such problems, requiring an answer more elaborate than "yes" or "no", *function problems*.

It is quite clear that decision problems are useful "surrogates" of function problems only in the context of *negative complexity results*. For example, now that we know that SAT and TSP (D) are **NP**-complete, we are sure that, unless $\mathbf{P} = \mathbf{NP}$, there is no polynomial algorithm for finding a satisfying truth assignment, or for finding the optimum tour. But, for all we know, some decision problems could be much easier than the original problems. We next point out that this is not the case for our two main examples.

Example 10.3: FSAT is the following function problem: Given a Boolean expression ϕ, if ϕ is satisfiable then return a satisfying truth assignment of ϕ; otherwise return "no". It calls for the computation of a *function* of sorts —hence the "F" in its name. For every input ϕ, this "function" may have many possible values (any satisfying truth assignment), or none (in which case we must return "no"); so it hardly conforms to our ideal of a mathematical function. However, it is a useful way for formalizing the actual computational task underlying SAT.

It is trivial that, if FSAT can be solved in polynomial time, then so can SAT. But it is not hard to see that the opposite implication also holds. If we had a polynomial algorithm for SAT, then we would also have a polynomial algorithm for FSAT, the following:

Given an expression ϕ with variables $x_1, \ldots, x_n$, our algorithm first asks whether ϕ is satisfiable. If the answer is "no," then the algorithm stops and returns "no." The difficult case is when the answer is "yes," and so our algorithm must come up with a satisfying truth assignment. The algorithm then does this: It considers the two expressions $\phi[x_1 = \mathbf{true}]$ and $\phi[x_1 = \mathbf{false}]$ resulting if x_1 is replaced by **true** and **false**, respectively. Obviously, at least one of them is satisfiable. Our algorithm decides whether $x_1 = \mathbf{true}$ or $x_1 = \mathbf{false}$ based on which one is satisfiable (if they both are, then we can break the tie any way we wish). Then the value of x_1 is substituted in ϕ, and x_2 is considered. It is clear that, with at most $2n$ calls of the assumed polynomial algorithm for satisfiability, all with expressions that are simpler than ϕ, our algorithm finds a satisfying truth assignment. □

The algorithm in the above example utilizes the interesting *self-reducibility* property of SAT and other **NP**-complete problems (self-reducibility is discussed again in Chapters 14 and 17). For the TSP, and other optimization problems, an extra trick is needed:

Example 10.4: We can solve the TSP (the function problem, in which the actual optimum tour must be returned) using a hypothetical algorithm for TSP (D), as follows: We first find the cost of the optimum tour by *binary search*. First notice that the optimum cost is an integer between 0 and 2^n, where n is the length of the encoding of the instance. Hence, we can find the precise cost by first asking whether, with the given cities and distances, budget 2^{n-1} is attainable. Next, depending on the answer, whether budget 2^{n-2} or $2^{n-2}+2^{n-3}$

is attainable, and so on, reducing the number of possible values for the optimum cost by half in each step.

Once the optimum cost is found, call it C, we shall ask TSP (D) questions in which the budget will always be C, but the intercity distances will change. In particular, we take all intercity distances one by one, and ask if the same instance, only with this particular intercity distance set to $C+1$, has a tour of cost C or less. If it does, then obviously the optimum tour does not use this intercity distance, and thus we may freeze its cost to $C+1$ with no harm. If, however, the answer is "no," then we know that the intercity distance under consideration is crucial to the optimum tour, and we restore it to its original value. It is easy to see that, after all n^2 intercity distances have been thus processed, the only entries of the distance matrix that are smaller than $C+1$ will be the n intercity links that are used in the optimum tour. All this requires polynomially many calls of the hypothetical algorithm for TSP (D). □

The relationship between decision and function problems can be formalized. Suppose that L is a language in **NP**. By Proposition 9.1, there is a polynomial-time decidable, polynomially balanced relation R_L such that for all strings x: There is a string y with $R_L(x,y)$ if and only if $x \in L$. The *function problem associated with* L, denoted FL, is the following computational problem:

Given x, find a string y such that $R_L(x,y)$ if such a string exists; if no such string exists, return "no".

The class of all function problems associated as above with languages in **NP** is called **FNP**. **FP** is the subclass resulting if we only consider function problems in **FNP** that can be solved in polynomial time. For example, FSAT is in **FNP**, but is not known (or expected) to be in **FP**; its special case FHORNSAT is in **FP**, and so is the problem of finding a matching in a bipartite graph. Notice that we have not claimed that TSP is a function problem in **FNP** (it probably is not, see Section 17.1). The reason is that, in the case of the TSP, the optimal solution is not an adequate certificate, as we do not know how to verify in polynomial time that it is optimal.

Definition 10.1: We can talk about reductions between function problems. We say that a function problem A reduces to function problem B if the following holds: There are string functions R and S, both computable in logarithmic space, such that for any strings x and z the following holds: If x is an instance of A then $R(x)$ is an instance of B. Furthermore, if z is a correct output of $R(x)$, then $S(z)$ is a correct output of x.

Notice the subtlety of the definition: R produces an instance $R(x)$ of the function problem B such that we can construct an output $S(z)$ for x from any correct output z of $R(x)$.

We say that a function problem A is complete for a class F$\mathcal{C}$ of function problems if it is in F$\mathcal{C}$, and all problems in that class reduce to A. □

There are no surprises: **FP** and **FNP** closed under reductions, and reductions among function problems compose. Furthermore, it is not hard to show that FSAT is **FNP**-complete. But we know, by the self-reducibility argument of Example 10.3, that FSAT can be solved in polynomial time if and only if SAT can. This establishes the following result:

Theorem 10.2: **FP** = **FNP** if and only if **P** = **NP**. □

Total Functions

The close parallels between function and decision problems, and especially Theorem 10.2, suggest that there is nothing exciting in the study of function problems in **FNP**. But there is one exception: There are certain important problems in **FNP** that are somehow guaranteed to never return "no". Obviously, such problems have no meaningful language or decision counterpart (they correspond to the trivial $L = \Sigma^*$). And despite this, they can be very intriguing and difficult computational problems, not known or believed to be in **FP**. We examine next some of the most important and representative examples.

Example 10.5: Consider the following famous function problem in **FNP**, called FACTORING: Given an integer N, find its prime decomposition $N = p_1^{k_1} p_2^{k_2} \ldots p_m^{k_m}$, *together with the primality certificates of* $p_1, \ldots, p_m$. Notice the requirement that the output includes the certificates of the prime divisors; without it the problem would not obviously be in **FNP**. Despite systematic and serious efforts for over two centuries, there is no known polynomial algorithm that solves this problem. It is plausible (although by no means universally believed) that there is no polynomial algorithm for FACTORING (but see Problem 10.4.11 for a promising approach).

Still, there are strong reasons to believe that FACTORING is very different from the other hard function problems in **FNP** (such as FSAT) that we have seen. The reason is that *it is a total function*. That is, for any integer N, such a decomposition is guaranteed to exist. In contrast, FSAT draws its difficulty precisely from the possibility that there may be no truth assignment satisfying the given expression. □

In general, we call a problem R in **FNP** *total* if for every string x there is at least one string y such that $R(x, y)$. The subclass of **FNP** containing all total function problems is denoted **TFNP**. Besides factoring, this class contains some other important problems that are not known to be in **FP**. We give three representative examples below:

Example 10.6: We are given an undirected graph (V, E) with integer (possibly negative) weights w on its edges (see Figure 10.1). Think of the nodes of the graph as people, and the weight on an edge an indication of how much (or little) these two people like each other — in a slight departure from reality, this

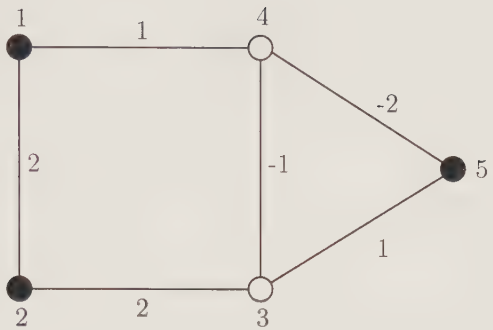


Figure 10-1. The HAPPYNET problem.

is assumed to be symmetric. A *state* S is a mapping from V to $\{-1, +1\}$, that is, an assignment to each node of one of the values $+1$ or -1. We say that node i is *happy* in state S if the following holds:

$$S(i) \cdot \sum_{[i,j] \in E} S(j) w[i,j] \geq 0. \tag{2}$$

Intuitively, condition (2) captures the fact that a node prefers to have the same value as an adjacent node to which it is connected via a positive edge, and the opposite value from a node adjacent via a negative edge. For example, in Figure 10.1 node 1 is happy, while node 3 is unhappy.

We can now define the following function problem, called HAPPYNET: Given a graph with weights, find a state in which all nodes are happy. At first, this problem seems like a typical hard combinatorial problem: Trying all states is out of the question, and there is no known polynomial-time algorithm for finding a happy state. But there is an important difference: *All instances of* HAPPYNET *are guaranteed to have a solution*, a state in which all nodes are happy.

Here is the proof: Consider the following "figure of merit" of state S:

$$\phi[S] = \sum_{[i,j] \in E} S(i) S(j) w[i,j]. \tag{3}$$

Suppose that node i is unhappy in S, that is,

$$S(i) \cdot \sum_{[i,j] \in E} S(j) \cdot w[i,j] = -\delta < 0. \tag{4}$$

Let S' now be the state which is identical to S, except that $S'(i) = -S(i)$ (we say that i was "flipped"), and consider $\phi[S']$. It should be clear by comparing (3) and (4) that $\phi[S'] = \phi[S] + 2\delta$. That is, the function ϕ is increased by at least two when we flip any unhappy node. But this suggests the following algorithm:

Start with any state S, and repeat:
While there is an unhappy node, flip it.

Since ϕ takes values in the range $[-W \ldots + W]$, where $W = \sum_{[i,j] \in E} |w[i,j]|$, and it is increased by two in each iteration, *this process must end up at a state with no unhappy nodes.* The proof is complete. It follows immediately that HAPPYNET is in the class **TFNP** of total function problems.

So, happy states always exist; the problem is how to find them. The iterative algorithm above is only "pseudopolynomial," because its time bound is proportional to the edge weights, not their logarithms (there are examples that show this algorithm is indeed exponential in the worst case, see the references in 10.4.17). Incidentally, HAPPYNET is equivalent the problem of finding stable states in neural networks in the Hopfield model (see the references in 10.4.17); despite this problem's practical importance, to date there are no known polynomial-time algorithms for solving it. □

Example 10.7: We know that it is **NP**-complete, given a graph, to find a Hamilton cycle. But what if a Hamilton cycle *is given*, and we are asked to find *another* Hamilton cycle? The existing cycle should surely facilitate our search for the new one. Unfortunately, it is not hard to see that even this problem, call it ANOTHER HAMILTON CYCLE, is **FNP**-complete (Problem 10.4.15).

But consider the same problem in a *cubic* graph — one with all degrees equal to three. It turns out that *if a cubic graph has a Hamilton cycle, then it must have a second one as well.*

The proof is this: Suppose that we are given a Hamilton cycle in a cubic graph, say $[1, 2, \ldots, n, 1]$. Delete edge $[1, 2]$ to obtain a Hamilton path (see Figure 10.2(a)). We shall only consider paths, such as the one in Figure 10.2(a), starting with node 1 and not using edge $[1, 2]$. Call any such Hamilton path a *candidate* (all paths in Figures 10.2(a) through 10.2(f) are candidates). Call two candidate paths *neighbors* if they have $n - 2$ edges in common (that is, all but one).

How many neighbors does a candidate path have? The answer depends on whether the other endpoint is node 2. If it is not, and the path is from 1 to $x \neq 2$, then we can obtain two distinct candidate neighbors by adding to the path each of the edges out of x that are not currently in the path (recall that the graph is cubic), and breaking the cycle in the unique way that yields

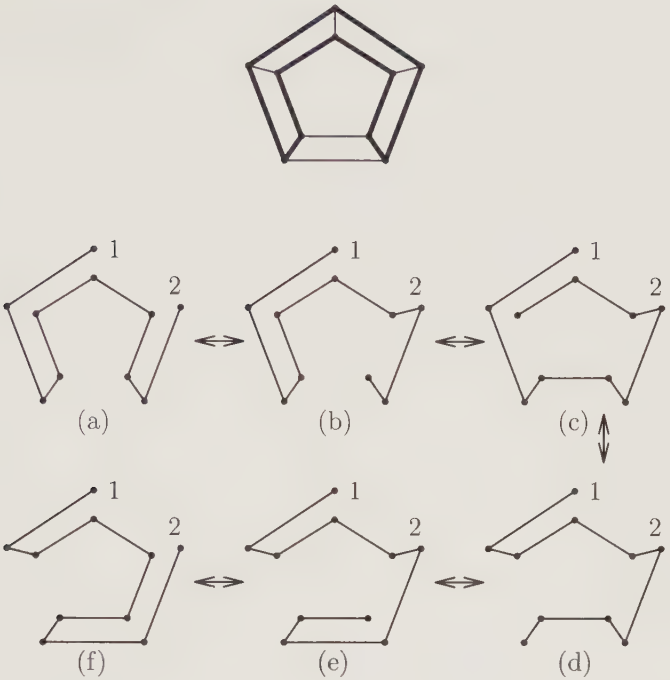


Figure 10-2. Finding ANOTHER HAMILTON CYCLE in a cubic graph.

another path. For example, the path in 10.2(c) has two neighbors, those in 10.2(b) and 10.2(d). But a candidate path with endpoints 1 and 2 has only one neighbor: Edge $[1, 2]$ cannot be used.

It is now obvious: Since all candidate paths have two neighbors except for those that have endpoints 1 and 2, which have only one neighbor, *there must be an even number of paths with endpoints* 1 *and* 2. But any Hamilton path from 1 to 2, with the addition of edge $[1, 2]$, will yield a Hamilton cycle. We conclude that *there is an even number of Hamilton cycles using edge* $[1, 2]$, and since we know of one, another must exist!

Again, there is no known polynomial-time algorithm for finding a second Hamilton cycle in a cubic graph. The algorithm suggested by the above argument and illustrated in Figure 10.2 (generate new neighbors until stuck) can be shown to be exponential in the worst case. However, the argument above does establish that ANOTHER HAMILTON CYCLE, for the special case of cubic graphs, is in **TFNP**. □

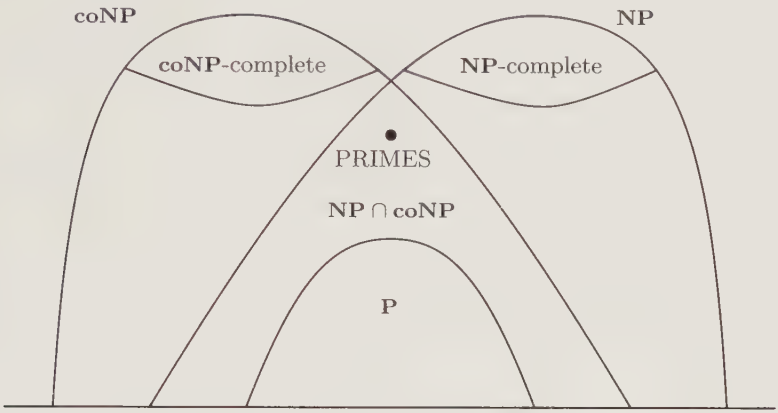
Example 10.8: Suppose that we are given n positive integers $a_1, \ldots, a_n$, such that $\sum_{i=1}^{n} a_i < 2^n - 1$. For example,

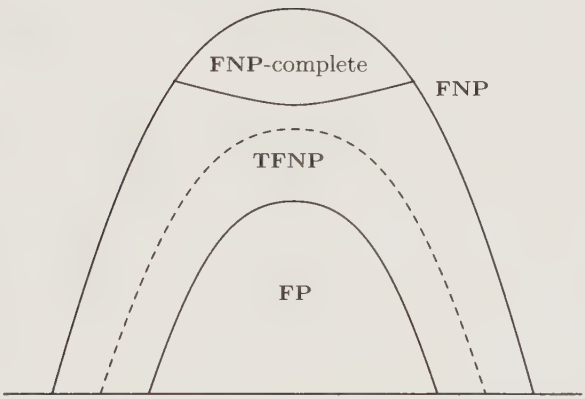
$$47, 59, 91, 100, 88, 111, 23, 133, 157, 205$$

(notice that their sum, 1014, is indeed less than $2^{10} - 1 = 1023$). Since there are more subsets of these numbers than there are numbers between 1 and $\sum_{i=1}^{n} a_i$, *there must be two different subsets which have the same sum.* In fact, it is easy to see that two *disjoint* subsets must exist that have the same sum. But there is no known polynomial-time algorithm for finding these two sets (the reader is invited to find the two sets in the above example). □

10.4 NOTES, REFERENCES, AND PROBLEMS

10.4.1 Class review.





10.4.2 As we mentioned in Note 8.4.1, the apparently stronger Karp reduction would not add much to our (or anybody's) list of **NP**-complete problems. But there is a different kind of reduction that does: The *polynomial-time nondeterministic reduction*, or γ-reduction, proposed in

- L. Adleman and K. Manders "Reducibility, randomness, and intractability," *Proc. 9th ACM Symp. on the Theory of Computing*, pp. 151–163, 1977.

In this reduction the output is computed by a polynomial-time bounded *nondeterministic* Turing machine. Recall what it means for a nondeterministic machine to compute a function, last discussed before the proof of Theorem 7.6: Although computations may give up, all other computations are correct (that is, the input is in L if and only if the output is in L'), and at least one computation does not give up. A problem is

called *γ-complete for* **NP** if it is in **NP** and all languages in **NP** γ-reduce to it.

Problem: Show that if L is γ-complete for **NP**, then $L \in \mathbf{P}$ if and only if $\mathbf{NP} = \mathbf{coNP}$.

There is at least one problem known to be γ-complete for **NP**, but not known to be **NP**-complete: LINEAR DIVISIBILITY, in which we are given two integers a and b, and we are asked whether there is an integer of the form $a \cdot x + 1$ that divides b, see the paper by Adleman and Manders cited above.

10.4.3 Problem: Show that **coNP** is the class of all graph-theoretic properties that can be expressed in *universal* second-order logic (recall Fagin's Theorem 8.3 for **NP**).

10.4.4 Recall the resolution method in Problem 4.4.10. We say that a set of clauses ϕ has *resolution depth* n if there is a sequence of sets of clauses $(\phi_0, \ldots, \phi_n)$ such that (a) $\phi_0 = \phi$; (b) ϕ_{i+1} is obtained from ϕ_i by adding to ϕ_i a resolvent of ϕ for $i = 0, \ldots, n-1$, and (c) ϕ_n contains the empty clause. We know from Problem 4.4.10 that all unsatisfiable expressions have some finite resolution depth. The *polynomial resolution conjecture* states that any unsatisfiable expression has resolution depth *polynomial* in the size of the expression.

Problem: Show that the polynomial resolution conjecture implies that $\mathbf{NP} = \mathbf{coNP}$.

The polynomial resolution conjecture was disproved in

- A. Haken "The intractability of resolution," *Theor. Comp. Sci 39*, pp. 297–308, 1985.

10.4.5 Duality. Let $L_1 = \{(x, K) :$ there is z such that $F_1(x, z)$ and $c_2(z) \geq K\}$ and $L_2 = \{(y, B) :$ there is z such that $F_2(y, z)$ and $c_2(z) \leq B\}$ be the decision versions of two optimization problems, one a minimization and the other a maximization problem, where the F_i's are polynomially balanced, polynomially computable relations, and the c_i are polynomially computable functions from strings to integers. Intuitively, $F_i(x, z)$ holds if and only if z is a feasible solution of the ith problem on input x, in which case $c_i(x, z)$ is the cost. Suppose that there are reductions from L_1 to $\overline{L}_2$, the complement of L_2, and back. We say that the two optimization problems are *dual* to each other.

Problem: (a) Show that, if L_1 and L_2 are decision versions of optimization problems that are dual to each other, then both languages are in $\mathbf{NP} \cap \mathbf{coNP}$.

(b) We are given the decision version of a minimization problem L_1 in terms of F_1 and c_1 as above. We say that L_1 has *optimality certificates* if the following language is in **NP**: $\{(x, z) :$ for all z', $F_1(x, z')$ implies $c_1(x, z) \leq c_1(x, z')\}$. Similarly for maximization problems. Show that the following statements are equivalent for the decision version L of an optimization problem:

(i) L has a dual.

(ii) L has optimality certificates.

(iii) L is in $\mathbf{NP} \cap \mathbf{coNP}$.

Duality is an important positive algorithmic property of optimization problems. In the case of MAX FLOW, its generalization LINEAR PROGRAMMING, and several

other important optimization problems, the duality consideration leads to some very elegant polynomial-time algorithms. In fact, duality is such a successful source of polynomial algorithms, that most known pairs of dual problems are in fact known to be in **P**. See

- C. H. Papadimitriou and K. Steiglitz *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, New Jersey, 1982, and
- M. Grötschel, L. Lovász, and A. Schrijver *Geometric Algorithms and Combinatorial Optimization*, Springer, Berlin, 1988.

10.4.6 Problem: A *strong* nondeterministic Turing machine is one that has three possible outcomes: "yes", "no", and "maybe." We say that such a machine decides L if this is true: If $x \in L$, then all computations end up with "yes" or "maybe," and at least one with "yes". If $x \notin L$, then all computations end up with "no" or "maybe," and at least one with "no". Show that L is decided by a strong nondeterministic machine if and only if $L \in \mathbf{NP} \cap \mathbf{coNP}$.

10.4.7 Problem: Show if x, y, and z are ℓ-bit integers, we can compute $x+y \bmod z$ and $x \cdot y \bmod z$ in time $\mathcal{O}(\ell^2)$. Show that we can compute $x^y \bmod z$ in time $\mathcal{O}(\ell^3)$.

10.4.8 Problem: Show that if N is not a prime, then it has a divisor other than one that is no larger than $\sqrt{N}$. Conclude that PRIMES can be solved in time $\mathcal{O}(\sqrt{N}\log^2 N)$ (where N is unfortunately *not* the length of the input, but the input itself).

10.4.9 Chinese remainder theorem, constructive version. Show that, if n is the product of distinct primes $p_1, \ldots, p_k$, the unique $r \in \Phi(n)$ such that $r_i = r \bmod p_i, i = 1, \ldots, k$ is given by the following formula: $r = \sum_{i=1}^{k} P_i Q_i r_i \bmod n$, where $P_i = \prod_{j \neq i} p_j$, and $Q_i = P_i^{-1} \bmod p_i$. (This is Langrange's Interpolation Formula transported to the modular domain. Show that the remainders are correct, and rely on the Chinese remainder theorem.)

10.4.10 Problem: Show that every odd prime square p^2 has a primitive root (Proposition 10.3). (Certainly p has a primitive root, call it r. Then either $r^{p-1} \neq 1 \bmod p^2$ or $(r+p)^{p-1} \neq 1 \bmod p^2$. Prove from this that either r or $r+p$ is a primitive root of p^2.)

10.4.11 Factoring algorithms. Although factoring integers is an ancient and honored problem, and number theorists never needed extra motivation to work on it, the advent of public-key cryptography (see Chapter 12) gave to this classical problem an unexpected practical significance. There are now algorithms for factoring large integers that are surprisingly fast, although superpolynomial in an intriguing way. We shall present the essence of a simple technique of this sort; for much more see

- A. K. Lenstra and H. W. Lenstra "Algorithms in number theory," *The Handbook of Theoretical Computer Science, vol. I: Algorithms and Complexity*, edited by J. van Leeuwen, MIT Press, Cambridge, Massachusetts, 1990

Fix the number n to be factored (the size of n will be the point of reference of our analysis). Call a number m *ℓ-smooth* if all prime factors of m are less than ℓ. Based

on the prime number theorem (stating that the number of primes less than n is about $\frac{n}{\ln n}$) one can calculate the probability that a randomly chosen number $m \leq n^\alpha$ is ℓ-smooth, where $\ell = e^{\beta\sqrt{\ln n \ln\ln n}}$.

(Incidentally, expressions such as this for ℓ are very common in this analysis; we shall write $\ell = L(\beta)$. The expected time of all such algorithms is also of this form, $L(\tau)$ for some τ; the effort is to minimize τ. In this context, we shall disregard factors that are $\mathcal{O}(L(\epsilon))$ for all $\epsilon > 0$, and thus do not affect τ.)

To continue the exposition, the probability that $m \leq n^\alpha$ is $L(\beta)$-smooth turns out to be $L(-\frac{\alpha}{2\beta})$. *Dixon's random squares algorithm* for factoring an integer n is this:

1. Randomly generate integers $m_1, m_2, \ldots \leq n$ until $L(\beta)$ m_i's are found such that $r_i = m_i^2 \bmod n$ is $L(\beta)$-smooth.
2. Find a subset S of these r_i's such that each prime $p \leq L(\beta)$ occurs an even number of times in S.

Once step 2 has been achieved, *we have found two numbers x and y such that* $x^2 = y^2 \bmod n$: x is the product of all m_i's such that $r_i \in S$; and y is the product of all primes less than $L(\beta)$, where each prime is raised to the power which is half the total number of times it appears in S.

(a) Show that indeed $x^2 = y^2 \bmod n$.

It can be shown that, with high probability, if $x^2 = y^2 \bmod n$, then the greatest common divisor of $x+y$ and n will be a nontrivial factor of n, and we have succeeded. The question is, how long does it take to achieve step 1, and how to achieve step 2.

(b) Using the probability of smoothness result quoted above, show that step 1 can be carried out in expected time $L(\beta + \frac{1}{2\beta})$, times the amount of time it takes us to test a number $\leq n$ for $L(\beta)$-smoothness. Show that this latter is $\mathcal{O}(L(\epsilon))$ for all $\epsilon > 0$ (and is therefore disregarded, recall our comment above).

(c) Show now that step 2 can be thought of as solving an $L(\beta) \times L(\beta)$ system of equations modulo 2; and thus it can be carried out in time $L(3\beta)$.

(d) Choose β so as to optimize the performance $L(\tau)$ of the overall algorithm.

10.4.12 Pratt's theorem is from

- V. R. Pratt "Every prime has a succinct certificate," *SIAM J. Comp.*, *4*, pp. 214–220, 1975.

For much more on number theory than the few elementary facts and techniques we develop in this chapter and the next, the reader is referred to

- G. H. Hardy and E. M. Wright *An Introduction to the Theory of Numbers*, Oxford Univ. Press, Oxford, U.K., 5th edition, 1979; and
- I. Niven and H. S. Zuckerman *An Introduction to the Theory of Numbers*, Wiley, New York 1972.

10.4.13 Problem: Show that FSAT is **FNP**-complete.

10.4.14 The *self-reducibility* of SAT exploited in Example 10.3 is a very important concept, to be seen again in this book (Sections 14.2 and 17.2). It was first pointed out in

- C. P. Schnorr "Optimal algorithms for self-reducible problems," *Proc. 3rd ICALP*, pp. 322-337, 1974.

10.4.15 Problem: Show that it is **NP**-complete, given a graph G and a Hamilton cycle of G, to decide whether G has *another* Hamilton cycle. (See the proof of Theorem 17.5.)

10.4.16 Problem: Each language $L \in \mathbf{NP} \cap \mathbf{coNP}$ suggests a function problem in **TFNP**. Which?

10.4.17 The class **TFNP** of total functions was first studied in

- N. Megiddo and C. H. Papadimitriou "On total functions, existence theorems, and computational complexity," *Theor. Comp. Sci., 81*, pp. 317–324, 1991.

The problem HAPPYNET (Example 10.6) is an interesting example of a function in **TFNP** which can be proved total by a "monotonicity" argument like that in equation (3) and thereafter. There is a subclass of **TFNP** called **PLS** (for *polynomial local search*) which contains many problems of this sort, including several **PLS**-complete ones—HAPPYNET is one of them. See

- D. S. Johnson, C. H. Papadimitriou, and M. Yannakakis "How easy is local search?" *Proc. 26th IEEE Symp. on the Foundations of Computer Science*, pp. 39–42, 1985; also, *J.CSS, 37*, pp. 79-100, 1988, and
- C. H. Papadimitriou, A. A. Schäffer, and M. Yannakakis "On the complexity of local search," *Proc. 22nd ACM Symp. on the Theory of Computing*, pp. , 838–445, 1990.

For more on neural networks see

- J. Hertz, A. Krog, and R. G. Palmer *Introduction to the Theory of Neural Computation*, Addison-Wesley, Reading, Massachusetts, 1991.

10.4.18 The problem of finding a second Hamilton cycle in a cubic graph (Example 10.7) exemplifies another genre of problems in **TFNP**, whose totality is based on the *parity argument*. Such problems form an intriguing complexity class that has interesting complete problems, see

- C. H. Papadimitriou "On graph-theoretic lemmata and complexity classes," *Proc. 31st IEEE Symp. on the Foundations of Computer Science*, pp. 794–801, 1990; retitled "On the complexity of the parity argument and other inefficient proofs of existence," to appear in *J.CSS*, 1993.

10.4.19 The problem of equal sums (Example 10.8) belongs to yet another class, since its totality is established by the "pigeonhole principle;" see the paper just referenced. A complete problem for this class is EQUAL OUTPUTS: Given a Boolean circuit C with n input gates and n output gates, either find an input that outputs $\mathbf{true}^n$, or find two different inputs that result in the same output. The equal sums problem is not known to be complete for this class.

Problem: (a) Show that the equal sums problem reduces to EQUAL OUTPUTS.

(b) Show that EQUAL OUTPUTS with *monotone* circuit C (that is, one without NOT gates) is in **P**.

CHAPTER 11

RANDOMIZED COMPUTATION

In some very real sense, computation is inherently randomized. It can be argued that the probability that a computer will be destroyed by a meteorite during any given microsecond of its operation is at least 2^{-100}.

11.1 RANDOMIZED ALGORITHMS

There are many computational problems for which the most natural algorithmic approach is based on *randomization*, that is, the hypothetical ability of algorithms to "flip unbiased coins." We examine several instances below.

Symbolic Determinants

In the bipartite matching problem (Section 1.2) we are given a bipartite graph $G = (U, V, E)$ with $U = \{u_1, \ldots, u_n\}$, $V = \{v_1, \ldots, v_n\}$, and $E \subseteq U \times V$, and we are asking whether there is a *perfect matching*, that is, a subset $M \subseteq E$ such that for any two edges (u, v) and (u', v') in M we have $u \neq u'$ and $v \neq v'$. In other words, we are seeking a *permutation* π of $\{1, \ldots, n\}$ such that $(u_i, v_{\pi(i)}) \in E$ for all $u_i \in U$.

There is an interesting way of looking at this problem in terms of matrices and determinants. Given such a bipartite graph G, consider the $n \times n$ matrix A^G whose i, jth element is a variable x_{ij} if $(u_i, v_j) \in E$, and it is zero otherwise. Consider next the *determinant* of A^G. Recall that it is defined as

$$\det A^G = \sum_{\pi} \sigma(\pi) \prod_{i=1}^{n} A^G_{i,\pi(i)},$$

where π ranges over all permutations of n elements, and $\sigma(\pi)$ is 1 if π is the product of an even number of transpositions, and -1 otherwise. It is clear that the only nonzero terms in this sum *are those that correspond to perfect matchings* π (this graph–theoretic view of determinants and permanents will be useful in several occasions). Furthermore, since all variables appear once, all of these terms are different monomials, and hence they do not cancel in the end result. It follows that *G has a matching if and only if* $\det A^G$ *is not identically zero.*

Thus, any algorithm for evaluating determinants of "symbolic matrices" such as $\det A^G$ would be an interesting alternative method for solving the matching problem. In fact, there are so many other applications of symbolic matrices (with entries that are general polynomials in several variables), that computing such a determinant, or just testing whether it is identically zero, is an important computational problem in its own right.

And we know how to compute determinants. The simplest and oldest method is *Gaussian elimination* (see Figure 11.1). We start by subtracting from rows $2, 3, \ldots, n$ appropriate multiples of the first row, so that all elements in the first column, except for the top one, become zero. Recall that such "elementary transformations" do not affect the determinant of the matrix. Then we consider the matrix that results if we disregard the first row and column, and repeat the same step. Naturally, at some point the "pivot" (the northwestern element of the unprocessed part of the matrix) may be zero (see the next to last stage in Figure 11.1). In this case, we rearrange the rows so as to correct this; if we cannot, the determinant is surely zero. Such interchange of columns multiplies the determinant by -1. Eventually, we shall end up with a matrix in "upper triangular form" (see the last matrix in the figure) The determinant of such a matrix is the product of the diagonal terms (in our example 196, and since there was one transposition of rows, the original determinant is -196). There is an issue that has to be settled before we proclaim this a polynomial algorithm: We must convince ourselves that the numerators and denominators of the elements created, if we start from an $n \times n$ matrix with entries integers with at most b bits, are not exponentially long in n and b. But this follows easily from the fact that these numbers are *subdeterminants* of the original matrix (Problem 11.5.3). We conclude that *we can calculate the determinant of a matrix in polynomial time.*

Can we apply this algorithm to symbolic matrices? If we try, the intermediate results are rational functions (see Figure 11.2). The reassuring fact we used in the numerical case (all intermediate results are ratios of subdeterminants of the original matrix, Problem 11.5.3) is now the source of concern, indeed an unsurmountable obstacle: These subdeterminants in general have exponentially many terms; see Figure 11.2 for a modest example. Even telling whether a specific monomial, like x^2zw, appears in the determinant with a non-

$$\begin{pmatrix} 1 & 3 & 2 & 5 \\ 1 & 7 & -2 & 4 \\ -1 & -3 & -2 & 2 \\ 0 & 1 & 6 & 2 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 3 & 2 & 5 \\ 0 & 4 & -4 & -1 \\ 0 & 0 & 0 & 7 \\ 0 & 1 & 6 & 2 \end{pmatrix} \Rightarrow$$

$$\begin{pmatrix} 1 & 3 & 2 & 5 \\ 0 & 4 & -4 & -1 \\ 0 & 0 & 0 & 7 \\ 0 & 0 & 7 & 2\frac{1}{4} \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 3 & 2 & 5 \\ 0 & 4 & -4 & -1 \\ 0 & 0 & 7 & 2\frac{1}{4} \\ 0 & 0 & 0 & 7 \end{pmatrix}$$

Figure 11.1. Gaussian elimination.

zero coefficient is **NP**-complete (Problem 11.5.4). Gaussian elimination seems to be of no help in evaluating symbolic determinants.

$$\begin{pmatrix} x & w & z \\ z & x & w \\ y & z & 0 \end{pmatrix} \Rightarrow \begin{pmatrix} x & w & z \\ 0 & \frac{x^2-zw}{x} & \frac{wx-z^2}{x} \\ 0 & \frac{zx-wy}{x} & -\frac{zy}{x} \end{pmatrix} \Rightarrow$$

$$\begin{pmatrix} x & w & z \\ 0 & \frac{x^2-zw}{x} & \frac{wx-z^2}{x} \\ 0 & 0 & -\frac{yz(xz-xw)+(zx-wy)(wx-z^2)}{x(x^2-zw)} \end{pmatrix}$$

Figure 11.2. Symbolic Gaussian elimination.

But remember that we are not interested in actually evaluating the symbolic determinant; we just need to tell whether it is identically zero or not. Here is an interesting idea: Suppose that we *substitute arbitrary integers for the variables*. Then we obtain a numerical matrix, whose determinant we can calculate in polynomial time by Gaussian elimination. If this determinant is not zero, then we *know* that the symbolic determinant we started with was not identically zero. And if the symbolic determinant is identically zero, our numerical result will always be zero. But we may be very unlucky, and the numbers we choose may be such that the numerical determinant is zero, although the symbolic one was not. In other words, we may stumble upon one of the *roots* of the determinant (seen as a polynomial). The following result reassures us that, with appropriate precautions, this is a very unlikely event:

Lemma 11.1: Let $\pi(x_1, \ldots, x_m)$ be a polynomial, not identically zero, in m variables each of degree at most d in it, and let $M > 0$ be an integer. Then the number of m-tuples $(x_1, \ldots, x_m) \in \{0, 1, \ldots, M-1\}^m$ such that $\pi(x_1, \ldots, x_m) = 0$ is at most mdM^{m-1}.

Proof: Induction on m, the number of variables. When $m = 1$ the lemma says that no polynomial of degree $\leq d$ can have more than d roots (the proof

of this well-known fact is identical to that of Lemma 10.4 for the mod p case). By induction, suppose the result is true for $m-1$ variables. Now write π as a polynomial in x_m, whose coefficients are polynomials in $x_1, \ldots, x_{m-1}$. Suppose that this polynomial evaluated at some integer point is zero. There are two cases: Either the highest-degree coefficient of x_m in π is zero, or it is not. Since this coefficient is a polynomial in $x_1, \ldots, x_{m-1}$, by induction the first case can occur for at most $(m-1)dM^{m-2}$ values of $x_1, \ldots, x_{m-1}$, and for each such value the polynomial will be zero for at most M values of x_m, that is, for $(m-1)dM^{m-1}$ values of $x_1, \ldots, x_m$ *in toto*. The second case defines a polynomial of degree $\leq d$ in x_m which can have at most d roots for each combination of values of $x_1, \ldots, x_{m-1}$, or dM^{m-1} new roots of π. Adding the two estimates we complete the proof. □

Lemma 11.1 suggests the following randomized algorithm for deciding if a graph G has a perfect matching. We denote by $A^G(x_1, \ldots, x_m)$ the matrix A^G with its m variables. Notice that $\det A^G(x_1, \ldots, x_m)$ has degree at most one in each of the variables.

Choose m random integers $i_1, \ldots, i_m$ between 0 and $M = 2m$.
Compute the determinant $\det A^G(i_1, \ldots, i_m)$ by Gaussian elimination.
If $\det A^G(i_1, \ldots, i_m) \neq 0$ then reply "G has a perfect matching"
If $\det A^G(i_1, \ldots, i_m) = 0$ then reply "G *probably* has no perfect matching."

We call the algorithm above a *polynomial Monte Carlo algorithm* for telling whether a bipartite graph has a perfect matching. By this we mean that, if the algorithm finds that a matching exists, its decision is reliable and final. But if the algorithm answers "probably no matching," then there is a possibility of a *false negative*. The point is that, if G does have a matching, the probability of a false negative answer is by Lemma 11.1 (and our choice of $M = 2m = 2md$) no more than one half. Notice that this is not a probabilistic statement over the space of all symbolic determinants or bipartite graphs; it is a probabilistic statement *about randomized computations*, and it holds *for all determinants and graphs*.

We already knew a deterministic polynomial algorithm for matching (recall Section 1.2); but this does not diminish the importance of the technique. For example, the same Monte Carlo algorithm obviously solves the more general problem of telling whether a symbolic matrix is *not* identically zero (for which no deterministic algorithm is known).

By taking M much larger than md we could reduce the probability of a false negative answer as much as desired (naturally, at the expense of applying Gaussian elimination to a matrix with larger numbers). However, there is a more appealing—and much more widely applicable— way of reducing the

chance of false negative answers: *Perform many independent experiments.* That is, if we repeat k times the evaluation of the determinant of a symbolic matrix, each time with independently chosen random integer values for the variables in the range $0 \ldots 2md - 1$, and the answer always comes out zero, then our confidence on the outcome that G has no perfect matching is boosted to $1-(\frac{1}{2})^k$. If the answer is different from zero even once, then we know that a perfect matching exists.

To summarize, in a Monte Carlo algorithm there can be no false positive answers, and the probability of false negatives is bounded away from one (say, less than one half). For the random choices, we have for the time being to assume the existence of a fair coin that generates perfectly random bits; we shall question this assumption in Section 11.3. Finally, the total time needed by the algorithm is always polynomial, for all possible random choices.

By studying algorithms whose outcome is less than perfectly reliable we are *not* dropping our standards of mathematical rigor and professional responsibility. Our estimate in the previous paragraph of the probability of a false negative answer is as rigorous as any mathematical statement. And nothing prevents us from running the Monte Carlo algorithm a hundred times, thus bringing its reliability far above that of other components of computation (not to mention life itself...).

Random Walks

Consider the following randomized algorithm for SAT:

> Start with any truth assignment T, and repeat the following r times:
> If there is no unsatisfied clause, then reply "formula is satisfiable" and halt.
> Otherwise, take any unsatisfied clause; all of its literals are **false** under T.
> *Pick any of these literals at random and flip it, updating* T.
> After r repetitions reply "formula is probably unsatisfiable."

We will fix parameter r later. Notice that we do not specify how an unsatisfied clause is picked, or how we choose the starting truth assignment —we are prepared to accept the worst case of these aspects of the algorithm. The only randomization we need is in choosing the literal to flip. We pick a literal among those in the chosen clause at random with equal probability. By "flipping it" we mean that the truth value of the corresponding variable is reversed in T. T is then updated, and the process repeated until either a satisfying truth assignment is discovered, or r flips have been performed. We call this *the random walk algorithm.*

If the given expression is unsatisfiable, then our algorithm is bound to be "correct": It will conclude that the expression is "probably unsatisfiable." But

what if the expression is satisfiable? Again, it is not hard to argue that, if we allow exponentially many repetitions we will eventually find a satisfying truth assignment with very high probability (see Problem 11.5.6).

The important question is, what are the chances that a satisfying truth assignment will be discovered when r is polynomial in the number of Boolean variables? Can this naive approach work against this formidable problem? Indeed not: There are simple satisfiable instances of 3SAT for which the "random walk algorithm" performs badly in all possible respects and metrics (Problem 11.5.6). But, interestingly, when applied to 2SAT the random walk algorithm performs quite decently:

Theorem 11.1: Suppose that the random walk algorithm with $r = 2n^2$ is applied to any satisfiable instance of 2SAT with n variables. Then the probability that a satisfying truth assignment will be discovered is at least $\frac{1}{2}$.

Proof: Let $\hat{T}$ be a truth assignment that satisfies the given 2SAT instance, and let $t(i)$ denote the expected number of repetitions of the flipping step until a satisfying truth assignment is found *assuming that our starting truth assignment T differs from $\hat{T}$ in exactly i values.* It is easy to see that this quantity is a finite function of i (Problem 11.5.5).

What do we know about $t(i)$? First of all, we know that $t(0) = 0$, since, if by a miracle we start at $\hat{T}$, then no flips will be necessary. Also we need not flip when we are at some other satisfying truth assignment. Otherwise, we must flip at least once. When we flip, we choose among the two literals of a clause that is not satisfied by the present T. Now, at least one of these two literals is **true** under $\hat{T}$—since $\hat{T}$ satisfies all clauses. Therefore, by flipping one randomly chosen literal, we have at least $\frac{1}{2}$ chance of moving closer to $\hat{T}$. Thus, for $0 < i < n$ we can write the inequality:

$$t(i) \leq \frac{1}{2}(t(i-1) + t(i+1)) + 1 \tag{3}$$

where the added unit stands for the flip just made. It is an inequality and not an equation, because the situation could be brighter: Perhaps the current T also satisfies the expression, or it differs from $\hat{T}$ in both literals, not just the guaranteed one. We also have $t(n) \leq t(n-1) + 1$, since at $i = n$ we can only decrease i.

Consider now another situation, where (3) holds *as an equation.* This way we give up the occasional chance of stumbling upon another satisfying truth assignment, or a clause where T and $\hat{T}$ differ in both literals. It is clear that this can only increase the $t(i)$'s. This means, that if we define the function $x(i)$ to obey $x(0) = 0$, $x(n) = x(n-1) + 1$ and $x(i) = \frac{1}{2}(x(i-1) + x(i+1)) + 1$, then we will have $x(i) \geq t(i)$ for all i.

Now the $x(i)$'s are easy to calculate. Technically, the situation is called a "one dimensional random walk with a reflecting and an absorbing barrier"

"gambler's ruin against the sheriff" is perhaps a little more graphic. If we add all equations on the $x(i)$'s together, we get $x(1) = 2n - 1$. Then solving the x_1–equation for x_2 we get $x_2 = 4n - 4$, and continuing like this $x(i) = 2in - i^2$. As expected, the worst starting i is n, with $x(n) = n^2$.

We have thus proved that the expected number of repetitions needed to discover a satisfying truth assignment is $t(i) \leq x(i) \leq x(n) = n^2$. That is, no matter where we start, our expected number of steps is at most n^2. The following useful lemma then, with $k = 2$, completes the proof of the theorem:

Lemma 11.2: If x is a random variable taking nonnegative integer values, then for any $k > 0$ $\mathbf{prob}[x \geq k \cdot \mathcal{E}(x)] \leq \frac{1}{k}$. ($\mathcal{E}(x)$ denotes the expected value of x.)

Proof: Let p_i be the probability that $x = i$. It is clear that

$$\mathcal{E}(x) = \sum_i ip_i = \sum_{i \leq k\mathcal{E}(x)} ip_i + \sum_{i > k\mathcal{E}(x)} ip_i > k\mathcal{E}(x)\mathbf{prob}[x > k \cdot \mathcal{E}(x)],$$

from which the lemma and (and the theorem) follow immediately. □

Theorem 11.1 implies that the random walk algorithm with $r = 2n^2$ is in fact a polynomial Monte Carlo algorithm for 2SAT. Once again there are no false positives, and the probability of a false negative is, by Lemma 11.2 with $k = 2$, less than $\frac{1}{2}$.

The Fermat Test

The two previous examples should not mislead the reader into believing that randomized algorithms only solve problems that we already knew how to solve. Testing whether a number is a prime is an interesting example in this regard.

We know from Lemma 10.3 that, if N is a prime, then, for all residues $a > 0$, $a^{N-1} = 1 \bmod N$. But suppose that N is not prime. What percentage of its residues have this property? Figure 11.3 shows, for each number between 2 and 20, the percentage of the nonzero residues that satisfy $a^{N-1} = 1 \bmod N$. A very tempting hypothesis emerges:

Hypothesis: If N is not a prime, then for at least half of its nonzero residues a we have $a^{N-1} \neq 1 \bmod N$.

This hypothesis immediately suggests a polynomial Monte Carlo algorithm for testing whether a number N is composite:

> Pick a random residue a modulo N.
> If $a^{N-1} \neq 1 \bmod N$ answer "N is composite."
> Otherwise answer "N is probably prime."

If the hypothesis is true, then the probability of false negatives is indeed less than one half. *Unfortunately, the hypothesis is false.* For example, all residues

2	100%
3	100%
4	33.3%
5	100%
6	20%
7	100%
8	14.3%
9	25%
10	11%
11	100%
12	9.1%
13	100%
14	7.7%
15	28.5%
16	13.3%
17	100%
18	6%
19	100%
20	5.3%
⋮	⋮
561	100%
⋮	⋮

Figure 11.3. How many pass the Fermat test.

of 561 pass the Fermat test for that number, and still $561 = 3 \times 11 \times 17$. The reason is that for all prime divisors p of $N = 561$, $p - 1 | N - 1$. A number N with this rare property is called *a Carmichael number;* Carmichael numbers seem to spoil this nice idea.

There are two clever ways for circumventing the problem of Carmichael numbers by making the Fermat test a little more sophisticated. We next turn to the development of some number-theoretic ideas that lead to one of them (for the other, which is in fact a little simpler, see the references in 11.5.7 and Problem 11.5.10).

Square Roots Modulo a Prime

We know that the equation $x^2 = a \bmod p$ has at most two roots (Lemma 10.4). It turns out that it has either two or none, and it is easy to tell which of the two cases pertains:

Lemma 11.3: If $a^{\frac{p-1}{2}} = 1 \bmod p$ then the equation $x^2 = a \bmod p$ has two roots. If $a^{\frac{p-1}{2}} \neq 1 \bmod p$ (and $a \neq 0$) then $a^{\frac{p-1}{2}} = -1 \bmod p$ and the equation $x^2 = a \bmod p$ has no roots.

Proof: Recall that, since p is a prime, it has a primitive root r. So, $a = r^i$ for some $i < p-1$. There are two cases: If $i = 2j$ is even, then obviously $a^{\frac{p-1}{2}} = r^{j(p-1)} = 1 \bmod p$, and a has two square roots: r^j and $r^{j+\frac{p-1}{2}}$. Notice that this accounts for half of the residues a, and since each has two square roots, we have run out of square roots! So, if $i = 2j+1$ is odd, then r^i can have no square roots, and $a^{\frac{p-1}{2}} = r^{\frac{p-1}{2}} \bmod p$. Now that latter number is a square root of 1, and it cannot be 1 itself (remember, r is a primitive root). So, it must be -1, the only other possible square root of one. □

So, the expression $a^{\frac{p-1}{2}} \bmod p \in \{1, -1\}$ is an important indicator of whether a is a perfect square modulo p or not. We abbreviate it as $(a|p)$ (where it is assumed that p is always a prime other than 2), pronounced *the Legendre symbol* of a and p. Notice immediately that $(ab|p) = (a|p)(b|p)$.

Suppose that p and q are two odd primes. We shall next discover another unexpected way of computing $(q|p)$:

Lemma 11.4 (Gauss's Lemma): $(q|p) = (-1)^m$, where m is the number of residues in the set $R = \{q \bmod p, 2q \bmod p, \ldots, \frac{p-1}{2}q \bmod p\}$ that are greater than $\frac{p-1}{2}$.

Proof: First all residues in R are distinct (if $aq = bq \bmod p$, then either $a - b$ or q divides p, absurd). Furthermore, no two elements of R add up to p (if $aq + bq = 0 \bmod p$ then either q or $a + b$ divides p; this is absurd since both a and b are at most $\frac{p-1}{2}$). Consider the set R' of residues that result from R if we replace each of the m elements $a \in R$ where $a > \frac{p-1}{2}$ by $p - a$ (see Figure 11.4 for an example with $p = 17$ and $q = 7$). Then all residues in R' are at most $\frac{p-1}{2}$. In fact, we claim that R' is precisely the set $\{1, 2, \ldots, \frac{p-1}{2}\}$ (otherwise, two elements of R would add up to p, which we have excluded).

Thus, modulo p, these two sets are the same: $\{1, 2, \ldots, \frac{p-1}{2}\}$; and $R' = \{\pm q, \pm 2q \ldots, \pm\frac{p-1}{2}q\}$ *where exactly* m *of the elements of the latter set have the minus sign.* Taking the product of all elements in the two sets modulo p, we have $\frac{p-1}{2}! = (-1)^m q^{\frac{p-1}{2}} \frac{p-1}{2}! \bmod p$. Since $\frac{p-1}{2}!$ cannot divide p, the lemma follows. □

We next show an important theorem that points out a subtle relationship between $(p|q)$ and $(q|p)$: They are the same, unless exactly one of p and q is -1 mod 4, in which case they are opposite.

Lemma 11.5 (Legendre's Law of Quadratic Reciprocity): $(p|q) \cdot (q|p) = (-1)^{\frac{p-1}{2}\frac{q-1}{2}}$.

Proof: Recall the set R' of the previous proof, and consider the sum of its

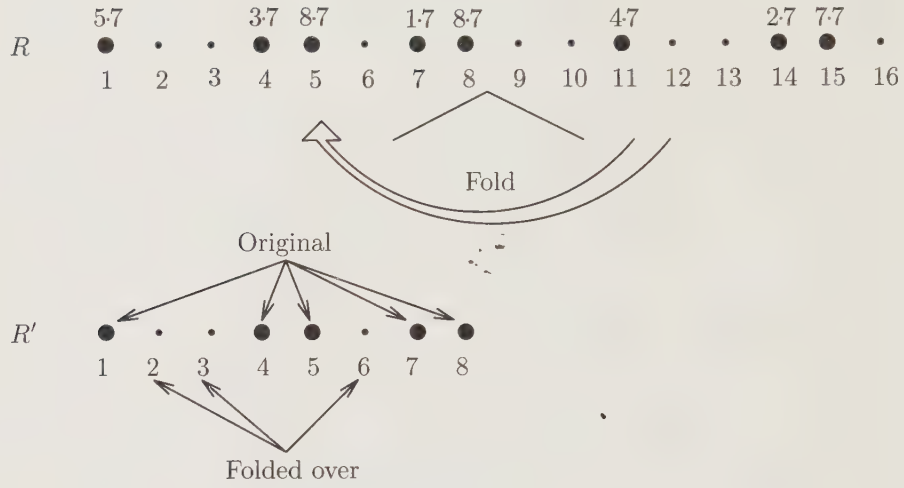


Figure 11-4. The sets R and R'.

elements modulo 2. Seen one way, it is just the sum of all integers between one and $\frac{p-1}{2}$, which is $\frac{(p-1)(p+1)}{2}$ mod 2. But seen another way, namely the way the set R' was derived, this sum is

$$q\sum_{i=1}^{\frac{p-1}{2}} i - p\sum_{i=1}^{\frac{p-1}{2}} \lfloor \frac{iq}{p} \rfloor + mp \bmod 2 \quad (1)$$

The first term is how we started, multiplying q by $1, 2, \ldots, \frac{p-1}{2}$. The second term reminds us that we took the residues of these numbers modulo p. And the last term, mp, notes the fact that we replaced m of these a's by the $p-a$'s. Naturally, (1) does not account for the reversal of the sign of these a's; however, since we are working modulo 2, reversals of sign do not affect the end result. Also because we are working modulo 2, we can omit from (1) factors of p and q (both odd primes). After all these simplifications, the first term of (1) becomes the same as our first estimate of $\frac{(p-1)(p+1)}{2}$ mod 2. Therefore we have:

$$m = \sum_{i=1}^{\frac{p-1}{2}} \lfloor \frac{iq}{p} \rfloor \bmod 2. \quad (2)$$

Now, the right-hand side of (2) can be easiest understood geometrically (see Figure 11.5 for $p = 17$, $q = 7$): It is the *number of positive integer points in the* $\frac{p-1}{2} \times \frac{q-1}{2}$ *rectangle that are under the line between* O *and the point* (p, q). By

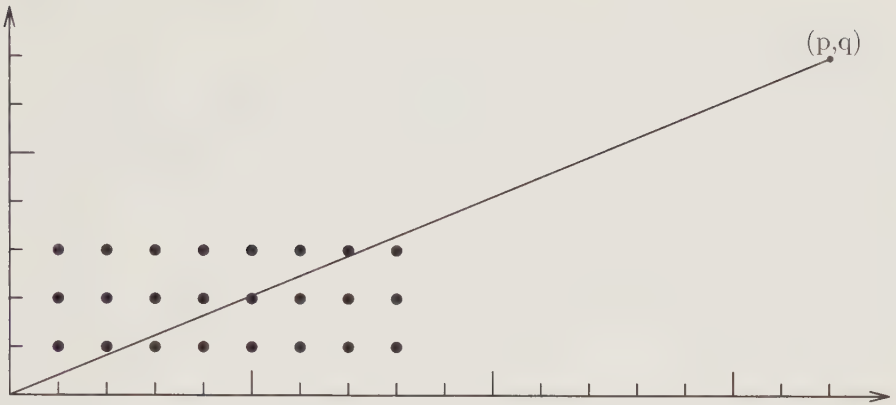


Figure 11-5. Eisenstein's rectangle.

Gauss's Lemma, equation (2) tells us that $(q|p)$ is precisely (-1) raised to this number.

The proof is almost complete. Repeat the same calculation, with the roles of p and q reversed. What we get is that $(p|q)$ is (-1) raised to the number of positive integer points in the $\frac{p-1}{2} \times \frac{q-1}{2}$ rectangle that are *above* the line. So, $(p|q) \cdot (q|p)$ is (-1) raised to the total number of integer points in the $\frac{p-1}{2} \times \frac{q-1}{2}$ rectangle, which is of course $\frac{p-1}{2}\frac{q-1}{2}$. □

In order to understand the significance of Lemma 11.5, we must extend the notation $(M|N)$ in cases in which M and N are not primes. Suppose that $N = q_1 \ldots q_n$, where the q_j's are all odd primes (not necessarily distinct). Then we define $(M|N)$ to be $\prod_{i=1}^{n}(M|q_i)$. This expression is now only a symbol, it has little to do with square roots. But it is a very useful symbol. Several properties that we observed for $(n|p)$ still hold for $(M|N)$. For example, it is multiplicative: $(M_1M_2|N) = (M_1|N)(M_2|N)$. Also, it is still a function of *residues*, that is $(M|N) = (M + N|N)$. What is more important, the Law of Quadratic Reciprocity still holds. We summarize these as follows:

Lemma 11.6: (a) $(M_1M_2|N) = (M_1|N)(M_2|N)$.

(b) $(M + N|N) = (M|N)$; that is, $(M|N)$ is a function of residues mod$N$.

(c) If M and N are odd, then $(N|M) \cdot (M|N) = (-1)^{\frac{M-1}{2}\frac{N-1}{2}}$.

Proof: (a) and (b) follow from the corresponding properties of $(p|q)$ by applying the definition of $(M|N)$. For (c), it suffices to observe that if a and b are odd numbers, then $\frac{a-1}{2}\frac{b-1}{2} = \frac{ab-1}{2}$ mod 2. □

These properties allow us to calculate $(M|N)$ without knowing the factorization of either M or N. The pattern is very similar to that of *Euclid's*

algorithm for computing the greatest common divisor (M, N) of two integers, so we recall this classical algorithm first.

Euclid's algorithm repeatedly replaces the larger of two integers by its residue modulo the larger one, until one of the numbers becomes zero—at which point, the other number is precisely the greatest common divisor sought. For example, here is how we compute the greatest common divisor of 51 and 91: $(51, 91) = (51, 40) = (11, 40) = (11, 7) = (4, 7) = (4, 3) = (1, 3) = (1, 0)$. It is easy to see that the largest of the two numbers is at least halved every two steps, and so the number of steps is at most 2ℓ, where ℓ is the number of bits in the two integers. Also, each division can be done in $\mathcal{O}(\ell^2)$ time (recall Problem 10.4.7).

$(M|N)$ can be computed in a very similar manner, using Lemma 11.6. One complication is that we cannot use the formula if $N = 2K$ is even, since $(M|N)$ is not defined for even N. We must compute $(2|M)$ from scratch, and then continue by $(2K|M) = (2|M)(K|M)$, etc. But $(2|M)$ is easy to compute: It is always $(-1)^{\frac{M^2-1}{8}}$ (Problem 11.5.9). For example, $(163|511) = -(511|163)$ by (c), $= -(22|163)$ by (b), $= -(2|163)(11|163)$ by (a), $= (11|163) = -(163|11)$ by (c), $= -(9|11)$ by (b) $= -(11|9)$ by (c), $= -(2|9) = -1$.

Lemma 11.7: Given two integers M and N, with $\lceil \log MN \rceil = \ell$, (M, N) and $(M|N)$ can be computed in $\mathcal{O}(\ell^3)$ time. □

We have already cautioned the reader that, if N is not an odd prime, then $(M|N)$ is just a notation. It has nothing to do with whether M is a perfect square modulo N, or with $M^{\frac{N-1}{2}} \bmod N$. In fact, if N is composite, then *at least half of the elements* M *of* $\Phi(N)$ *satisfy* $(M|N) \neq M^{\frac{N-1}{2}} \bmod N$. This suggests a Monte Carlo test for compositeness, and makes $(M|N)$ an interesting quantity to know how to calculate. We prove something weaker first:

Lemma 11.8: If $(M|N) = M^{\frac{N-1}{2}} \bmod N$ for all $M \in \Phi(N)$, then N is a prime.

Proof: For the sake of contradiction, suppose that $(M|N) = M^{\frac{N-1}{2}} \bmod N$ for all $M \in \Phi(N)$, and still N is composite. First suppose that N is the product of distinct primes $p_1, \ldots, p_k$. Let $r \in \Phi(p_1)$ have $(r|p_1) = -1$. By the Chinese remainder theorem (Corollary 2 to Lemma 10.2), there is an $M \in \Phi(n)$ such that $M = r \bmod p_1$, and $M = 1 \bmod p_i$, $i = 2, \ldots, k$. By the hypothesis, $M^{\frac{N-1}{2}} = (M|N) = -1 \bmod N$. But taking this last equation modulo p_2 we arrive at the contradiction $1 = -1 \bmod p_2$ (remember that, since we are discussing $(M|N)$, 2 is not a factor of N).

So, we must assume that $N = p^2 m$ for some prime $p > 2$ and integer m. Then let r be a primitive root modulo p^2 (recall Proposition 10.3). We claim that $r^{N-1} \neq 1 \bmod N$ (and this will contradict our hypothesis). In proof, if this were the case, then $N - 1$ must be a multiple of $\phi(p^2) = p(p-1)$, and so

p divides both N and $N-1$, absurd. □

From this it is easy to derive the main result:

Theorem 11.2: If N is an odd composite, then for at least half of $M \in \Phi(N)$, $(M|N) \neq M^{\frac{N-1}{2}} \bmod N$.

Proof: By Lemma 11.8, there is at least one $a \in \Phi(N)$ such that $(a|N) \neq a^{\frac{N-1}{2}} \bmod N$. Let $B = \{b_1, \ldots, b_k\} \subseteq \Phi(N)$ be the set of all distinct residues such that $(b_i|N) = b_i^{\frac{N-1}{2}} \bmod N$. Consider now the set $a \cdot B = \{ab_1 \bmod N, \ldots, ab_k \bmod N\}^{\dagger}$. We claim that all residues in $a \cdot B$ are distinct, and different from all residues in B (notice that this would conclude the proof). They are distinct because if $ab_i = ab_j \bmod N$ for $i \neq j$, then $a|b_i - b_j|$ divides N, absurd since $a \in \Phi(N)$ and $|b_i - b_j|$ is smaller than N. And they are not in B, because $(ab_i)^{\frac{N-1}{2}} = a^{\frac{N-1}{2}} b_i^{\frac{N-1}{2}} \neq (a|N)(b_i|N) = (ab_i|N)$. □

Corollary: There is a polynomial Monte Carlo algorithm for compositeness.

Proof: Given N, an odd integer, the algorithm is this:

> Generate a random integer M between 2 and $N-1$, and calculate (M, N).
> If $(M, N) > 1$, then conclude "N is a composite."
> Otherwise, calculate $(M|N)$ and $M^{\frac{N-1}{2}} \bmod N$, and compare the results;
> if they are not equal then reply "N is a composite,"
> otherwise reply "N is probably a prime."

(Incidentally, notice that any M that passes this test also passes the Fermat test for N, but obviously not vice–versa.) It is clear that there can be no false positives (the algorithm never mistakes a prime for a composite), and it follows from Theorem 11.2 that the probability of a false negative (the algorithm mistaking a composite for a prime) is at most $\frac{1}{2}$. By Lemma 11.7, all steps can be performed in time polynomial in the number of bits of N. The corollary follows. □

11.2 RANDOMIZED COMPLEXITY CLASSES

In order to study Monte Carlo algorithms formally, it would appear that we must introduce to our Turing machines a "coin-flipping" feature. Nothing of the sort is necessary. We shall model randomized algorithms as ordinary nondeterministic Turing machines, only with *a different interpretation of what it means for such a machine to accept its input.*

---

† Notice that what we are arguing here is that B is a proper subgroup of $\Phi(N)$; end of proof by group theory!

Definition 11.1: Let N be a polynomial-time bounded nondeterministic Turing machine. We can assume that N is *precise*, that is, all of its computations on input x halt after the same number of steps, which is a polynomial in $|x|$ (recall Proposition 7.1). We also assume that at each step there are exactly two nondeterministic choices for the machine (recall Figure 8.5).

Let L be a language. A *polynomial Monte Carlo Turing machine for* L is a nondeterministic Turing machine, standardized as above, with a number of steps in each computation on an input of length n which is a polynomial in n, call it $p(n)$, and such that the following is true for each string x: If $x \in L$, then *at least half* of the $2^{p(|x|)}$ computations of N on x halt with "yes". If $x \notin L$, then *all* computations halt with "no". The class of all languages with polynomial Monte Carlo Turing machines is denoted **RP** (for *randomized polynomial time*). □

Notice immediately that this definition captures our informal notion of a Monte Carlo algorithm. All nondeterministic steps are "coin flips." Naturally, in practice not all steps of a Monte Carlo algorithm are random choices, but there is no harm in assuming that a coin is flipped at each step, and its outcome is ignored most of the time (that is, the two alternatives are identical). Also, random choices with outcomes more complex than mere bits (for example, picking an integer between 1 and N in the Monte Carlo algorithm for compositeness) can be further broken down to binary ones (flipping the first bit of the integer, then the second, etc., rejecting if an integer outside the appropriate range results). There can be no false positive answers, since for $x \notin L$ rejection is unanimous. Also, the probability of false negatives is at most $\frac{1}{2}$; the reason is that, if each step of the machine involves a random choice between the two alternatives, with probability $\frac{1}{2}$ for each, then all computations, or "leaves" of the computation tree, are equiprobable events, each with probability $2^{-p(|x|)}$. Therefore, the requirement that at most half of them reject guarantees that the probability of a false negative is at most one half.

The power of **RP** would not be affected if the probability of acceptance were not $\frac{1}{2}$, as we currently require, but any number strictly between zero and one. The reason is this: If we have a randomized algorithm with probability of false negatives at most $1-\epsilon$, for some $\epsilon < \frac{1}{2}$, then we could transform it into one with probability at most $\frac{1}{2}$ by repeating the algorithm k times. "Repeating" in the Turing machine domain means that from each leaf of the nondeterministic computation tree we "hang" another tree identical to the original one. And so on for k repetitions. At the final leaves we report "yes" if and only if at least one computation leading to this leaf has reported "yes". The probability of a false negative answer is now at most $(1-\epsilon)^k$. By taking $k = \lceil -\frac{1}{\log(1-\epsilon)} \rceil$ we make sure that this probability is at least $\frac{1}{2}$. The total time required is k times the original polynomial. Notice here that ϵ can be an arbitrarily small constant; it

could even be a function of the form $\frac{1}{p(n)}$ *as long as* $p(n)$ *is a polynomial* (recall that $-\frac{1}{\log(1-\epsilon)} \approx \frac{1}{\epsilon}$). Similarly, by independent repetitions of a Monte Carlo algorithm we can make the probability of false negatives arbitrarily small—up to an inverse *exponential*.

Where does this new class **RP** fit in the realm of classes we have seen so far? It is clear that **RP** lies somewhere between **P** and **NP** (naturally, for all we know **P** may be equal to **NP**, and so there is nothing "in between"). To see why, notice that a polynomial Monte Carlo algorithm for L is by definition a polynomial nondeterministic machine deciding L (since at least half of the computations accept a string in L, surely at least one does). Also, a polynomial deterministic algorithm is a special case of a Monte Carlo algorithm: It is one that always ignores the coin flips and accepts unanimously (probability 1 is certainly at least as large as $\frac{1}{2}$).

RP is in some sense a new and unusual kind of complexity class. Not any polynomially bounded nondeterministic Turing machine can be the basis of defining a language in **RP**. For a machine N to define a language in **RP**, it must have the remarkable property that on all inputs it either rejects "unanimously," or it accepts "by majority." Most nondeterministic machines behave in other ways for at least some inputs. And given a machine, there is no easy way to tell whether it qualifies (in fact, this is an undecidable problem, see Problem 11.5.12). Finally, there is no easy way to *standardize* nondeterministic Turing machines so that the Monte Carlo property is self-evident, and still all Monte Carlo algorithms are included (as we did for properties like "halts within time n^3" by adopting the standard precise machines and their alarm clocks). There is a similar problem with the classes $\mathbf{NP} \cap \mathbf{coNP}$ and **TFNP** introduced in the previous chapter: There is no easy way to tell whether a machine always halts with a certified output. We informally call such classes *semantic* classes, as opposed to the *syntactic* classes such as **P** and **NP**, where we can tell immediately by a superficial check whether an appropriately standardized machine indeed defines a language in the class. One of the shortcomings of semantic classes is that, generally, there are no known complete problems for them[†]. The difficulty is this: Any syntactic class has a "standard" complete language, namely

$$\{(M, x) : M \in \mathcal{M} \text{ and } M(x) = \text{"yes"}\}$$

where $\mathcal{M}$ is the class of all machines of the variant that define the class, appropriately standardized (as long as the time bounds defining the class are all polynomials or another suitable class of functions). The proof of Cook's theorem, for example, is in essence a reduction from this language to SAT. In the

[†] In fact, the absence of complete problems is a perfectly good way of formalizing what we mean by a "semantic" class; see also Problem 20.2.14.

case of semantic classes, however, the "standard" complete language is usually undecidable!

The Class ZPP

Is **RP** a subset of **NP** ∩ **coNP**? Or, more ambitiously, is **RP** closed under complement? The asymmetry in the definition of **RP**, reminiscent of the asymmetry in the definition of **NP**, is quite alarming in this respect. Thus, there is a class **coRP** of problems that have a Monte Carlo algorithm with a limited number of false positives but no false negatives; for example, we know from the previous section that PRIMES is in this class. In some sense, **coRP** shares with **RP** the possibility that a false answer (false positive in this case) will survive our repeated trials. And, as in **RP**, there is no way of telling when we have tried enough independent experiments.

In this context, the class **RP** ∩ **coRP** seems very attractive (recall our discussion of **NP** ∩ **coNP** in the previous chapter). A problem in this class possesses two Monte Carlo algorithms: One that has no false positives, and one with no false negatives. Hence, if we run many independent experiments *with both algorithms*, sooner or later a definitive answer will come—either a positive answer from the algorithm with no false positives, or a negative one from the algorithm with no false negatives. If we execute both algorithms independently k times, the probability that no definitive answer is obtained falls to 2^{-k}. The difference with ordinary Monte Carlo algorithms is that in the end *we will know the correct answer for sure.* Of course, we are not sure *a priori* when the algorithm will end –although it is unthinkable that we will have no definite answer after one hundred repetitions. Such algorithms are called *Las Vegas algorithms* (perhaps to emphasize the fact that the algorithm's proprietor cannot lose).

The class **RP** ∩ **coRP** of languages with Las Vegas algorithms is denoted **ZPP** (for polynomial randomized algorithms with *zero probability* of error). As it turns out, PRIMES is in **ZPP** (see the references in 11.5.7).

The Class PP

Consider the problem MAJSAT: Given a Boolean expression, is it true that the majority of the 2^n truth assignments to its variables (that is, at least $2^{n-1}+1$ of them) satisfy it? It is not clear at all that this problem is in **NP**: The obvious certificate, consisting of of $2^{n-1}+1$ satisfying truth assignments, is not succinct at all. Naturally, MAJSAT is even less likely to be in **RP**.

There is a complexity class that is very appropriate for this problem: We say that language L is in the class **PP** if there is a nondeterministic polynomially bounded Turing machine N (standardized as above) such that, for all inputs x, $x \in L$ if and only if *more than half of the computations of N on input x end*

up accepting. We say that N decides L "by majority."

Notice that **PP** is a "syntactic," and not a "semantic," class. Any nondeterministic polynomially bounded Turing machine can be used to define a language in **PP**; no special properties are required. As a result, it has complete problems: It should be clear that MAJSAT is **PP**-complete (Problem 11.5.16). We can show the following:

Theorem 11.3: $\mathbf{NP} \subseteq \mathbf{PP}$.

Proof: Suppose that $L \in \mathbf{NP}$ is decided by a nondeterministic machine N. The following machine N' then will decide L by majority: N' is identical to N except that it has a new initial state, and a nondeterministic choice out of its initial state. One of the two possible moves gets us to the ordinary computation of N with the same input. The other choice gets us to a computation (with the same number of steps) that always accepts.

Consider a string x. If N on x computes for $p(|x|)$ steps and produces $2^{p(|x|)}$ computations, N' obviously will have $2^{p(|x|)+1}$ computations. Of these, at least half will halt with "yes" (the ones corresponding to the half of the computations of N' that accept unconditionally). Thus, a majority of the computations of N' accept if and only if there is at least one computation of N on x that accepts; that is, if and only if $x \in L$. Hence N' accepts L by majority, and $L \in \mathbf{PP}$. □

Is **PP** closed under complement? The only asymmetry between "yes" and "no" is the possibility of a "split vote," where there is an equal number of "yes" and "no" computations; but this is easily taken care of (see Problem 11.5.17).

The class BPP

Despite the fact that all three classes **RP**, **ZPP**, and **PP** are motivated by a probabilistic interpretation of nondeterministic choices, there is a big difference between the former two and the latter. **RP** and **ZPP** are *plausible notions of efficient randomized computation*, realistic proposals for practical algorithms; in this sense they are relatives of **P**. In contrast, **PP** is a natural way of capturing certain computational problems such as MAJSAT, but has no realistic computational content; in this sense it is closer to **NP**.

The reason that there is no direct way to exploit **PP** algorithmically is that acceptance by majority is an input-output convention that is "too fragile." A string x may be in L with an acceptance probability of $\frac{1}{2} + 2^{-p(|x|)}$, with just two more accepting computations than rejecting computations. And *there is no plausible efficient experimentation* that can detect such marginally accepting behavior.

To understand the last statement, imagine the following situation. You have a coin that is biased, that is, one of its sides is more likely to appear than the other. You know one side has probability $1 + \epsilon$, for some $\epsilon > 0$, and the other $1 - \epsilon$, *but you do not know which is which*. How would you detect which

side is the more likely? The obvious experimentation would be to flip the coin many times, and pick the side that appeared the most times to be the one with probability $1+\epsilon$. The question is, how many times do you have to flip in order to be able to guess correctly with reasonably high probability? The following result is most helpful in analyzing randomized algorithms:

Lemma 11.9 (The Chernoff Bound): Suppose that $x_1, \ldots, x_n$ are independent random variables taking the values 1 and 0 with probabilities p and $1-p$, respectively, and consider their sum $X = \sum_{i=1}^{n} x_i$. Then for all $0 \leq \theta \leq 1$, $\mathbf{prob}[X \geq (1+\theta)pn] \leq e^{-\frac{\theta^2}{3}pn}$.

Proof: If t is any positive real number, we have trivially that $\mathbf{prob}[X \geq (1+\theta)pn] = \mathbf{prob}[e^{tX} \geq e^{t(1+\theta)pn}]$. Recall from Lemma 11.2 that $\mathbf{prob}[e^{tX} \geq k\mathcal{E}(e^{tX})] \leq \frac{1}{k}$ for any real $k > 0$ (strictly speaking, that result was proved for integer random variables and integer k, but the proof is the same for the general case, with integrals replacing sums). Taking $k = e^{t(1+\theta)pn}[\mathcal{E}(e^{tX})]^{-1}$ we obtain

$$\mathbf{prob}[X \geq (1+\theta)pn] \leq e^{-t(1+\theta)pn}\mathcal{E}(e^{tX}).$$

Since $X = \sum_{i=1}^{n} x_i$, we have that $\mathcal{E}(e^{tX}) = (\mathcal{E}(e^{tx_1}))^n = (1+p(e^t-1))^n$. Substituting we get

$$\mathbf{prob}[X \geq (1+\theta)pn] \leq e^{-t(1+\theta)pn}(1+p(e^t-1))^n \leq e^{-t(1+\theta)pn}e^{pn(e^t-1)},$$

the last inequality because, for all positive a, $(1+a)^n \leq e^{an}$. Taking now $t = \ln(1+\theta)$ we get that

$$\mathbf{prob}[X \geq (1+\theta)pn] \leq e^{pn(\theta-(1+\theta)\ln(1+\theta))}.$$

Since the exponent expands to $-\frac{1}{2}\theta^2 + \frac{1}{6}\theta^3 - \frac{1}{12}\theta^4 + \ldots$, for $0 \leq \theta \leq 1$, the lemma follows. □

In other words, the probability that a *Bernoulli random variable* (as X is called) deviates from its expectation decreases exponentially with the deviation. This useful result can be now specialized as follows:

Corollary: If $p = \frac{1}{2} + \epsilon$ for some $\epsilon > 0$, then the probability that $\sum_{i=1}^{n} x_i \leq \frac{n}{2}$ is at most $e^{-\frac{\epsilon^2 n}{6}}$.

Proof: Take $\theta = \frac{\epsilon}{\frac{1}{2}+\epsilon}$. □

We can therefore detect in our coin a bias ϵ with reasonable confidence by taking the majority of about $\frac{1}{\epsilon^2}$ experiments. **PP** on the other hand, where the bias ϵ can be as small as $2^{-p(n)}$, is an inappropriate randomized complexity class: An *exponential* number of repetitions of the algorithm is required to determine the correct answer with reasonable confidence.

Definition 11.2: We next introduce perhaps the most comprehensive yet plausible notion of realistic computation that has been proposed. The class **BPP** contains all languages L for which there is a nondeterministic polynomially bounded Turing machine N (whose computations are all of the same length, as usual) with the following property: For all inputs x, if $x \in L$ then at least $\frac{3}{4}$ of the computations of N on x accept; and if $x \notin L$ then at least $\frac{3}{4}$ of them reject. □

That is, we require that N accept by a "clear majority," or reject by a "clear majority." The number $\frac{3}{4}$ in this definition is indicative of what is needed, namely to *bound the probability of a right answer away from half by a decent amount* (**BPP** stands for "bounded probability of error"). Any number strictly between $\frac{1}{2}$ and 1 would result in the same class. To see this, suppose that we have a machine N that decides L by majority $\frac{1}{2} + \epsilon$. We can run the machine $2k + 1$ times (by "hanging computations" from each leaf as in the proof of Proposition 11.3) and accept as outcome the majority of the outcomes. According to Lemma 11.9, the probability of a false answer is at most $e^{-2\epsilon^2 k}$, which can be made arbitrarily small by appropriately increasing k. In particular, by taking $k = \lceil \frac{\ln 2}{\epsilon^2} \rceil$ we achieve a probability of error at most $\frac{1}{4}$. Notice again that ϵ need not be a constant; it could be any *inverse polynomial*.

It should be clear that **RP** $\subseteq$ **BPP** $\subseteq$ **PP**. The first inclusion holds because any language in **RP** has a **BPP** algorithm: Just run the algorithm twice, to assure that the probability of false negatives is less than $\frac{1}{4}$ (the probability of false positives is zero, and thus already less than $\frac{1}{4}$). Finally, any language in **BPP** is also in **PP**, since a machine that decides by clear majority certainly decides by simple majority.

It is open whether **BPP** $\subseteq$ **NP**; but see Problem 11.5.18 and Section 17.2 for interesting results in this regard. Also, notice that the definition of **BPP** is symmetric: Acceptance by clear majority of "yes", rejection by clear majority of "no". Hence **BPP** is closed under complement, and **BPP** = **coBPP**. Notice finally that **BPP** is a "semantic" class: For a nondeterministic machine to define a language in **BPP**, it must have the property that for all inputs one of the two outcomes has a clear majority, one that is not obvious how to check, or standardize.

11.3 RANDOM SOURCES

Although **RP** and **BPP** are perfectly well–defined complexity classes, their practical relevance and importance rests on the hypothesis that we can implement randomized algorithms; in other words, that *we have a source of random bits.*

To formalize what is ideally needed, let us define a *perfect random source* to be a random variable with values that are infinite sequences $(x_1, x_2, \ldots)$ of

bits such that for all $n > 0$ and for all $(y_1, \ldots, y_n) \in \{0,1\}^n$ we have

$$\mathbf{prob}[x_i = y_i, i = 1, \ldots, n] = 2^{-n}.$$

That is, the x_i's are the outcomes of independent experiments, where each x_i is one with probability $p = \frac{1}{2}$.

If we had a perfect random source (that is, a physical device which, upon pushing a button, would start generating such a sequence $x_1, x_2, x_3 \ldots$) then we could *implement* any Monte Carlo algorithm on any input by simulating the corresponding nondeterministic machine, choosing the appropriate transition at step i according to x_i. (Actually, this is precisely how Monte Carlo algorithms were introduced informally in Section 11.1.) Also, any randomized algorithm for a language in **BPP** can be similarly implemented. Thus, given a perfect random source, problems in these complexity classes can be realistically solved in a satisfactory way.

But are there perfect random sources in nature? There are some plausible physical sources of high–quality random bits, but all are arguably flawed as perfect random sources. There is serious doubt whether a truly perfect source is physically possible (the reader is encouraged to think of a few alternatives, and their weaknesses).

Example 11.1: A perfect random source must have both *independence* (the probability that $x_i = 1$ should not depend on the previous, or future, outcomes), as well as *fairness* (the probability should be exactly $\frac{1}{2}$).

It turns out that the important requirement is independence: As John von Neumann observed a long time ago, we can turn an independent but unfair random source into a perfect one very easily: Just break the sequence into pairs, interpret 01 as 0, 10 as 1, *and ignore pairs* 00 *and* 11. That is, the sequence 001010000100011110... is broken into $(00, 10, 10, 00, 01, 00, 01, 11, 10, \ldots)$, and interpreted as 11001... The resulting random source is perfect.

Notice that we need not know the precise probability p that the outcome be 1, as long as it is strictly between zero and one, and it remains constant from one toss to the next. Also, to get a perfect random sequence of length n with this scheme we will need a sequence from the given source of expected length $\frac{2}{1-c}$, where $c = p^2 + (1-p)^2$ is the *coincidence probability* of the source, that is, the probability that the outcomes of two independent experiments will coincide. We shall see this quantity again later in this section. □

The real problem with physically implementing perfect random sources is that any physical process tends to be affected by its previous outcome (and the circumstances that led to it). This dependence may become weaker as the time between two consecutive experiments increases, but theoretically it never goes away.

Given the difficulties in implementing perfect random sources physically, we may try to discover randomness not in physical processes, but in mathematical

and computational ones. This brings us to the so called *pseudorandom number generators*, algorithms that produce sequences of bits that are in some sense "unpredictable" or "random." An elegant theory of pseudorandomness has been recently developed that explores this possibility rigorously, based on complexity and cryptography (see the references in 11.5.21). On the other hand, actual pseudorandom number generators found in computer systems often start with a provided *seed* which is several bits long (we can consider such a seed as a positive integer x_0), and then generate a sequence of integers like this: $x_{i+1} = ax_i + b \bmod c$, where a, b, and c are fixed integers. Unfortunately, seen in the light of the complexity theory of pseudorandomness, all such generators are provably *terrible* (see the discussion in 11.5.21).

Slightly Random Sources

Since the notion of perfect random sources seems not promising at all in terms of physical implementation, we are motivated to look at a weaker concept of randomness, which turns out to be much more plausible physically. Let δ be a number such that $0 < \delta \leq \frac{1}{2}$, and let p be any function mapping $\{0,1\}^*$ to the interval $[\delta, 1-\delta]$. The intention is that p is a highly complex function, completely unknown to us. The *δ–random source* S_p is again a random variable with infinite bit sequences as values, where the probability that the first n bits have the specific values $y_1, \ldots, y_n$ is now given by

$$\prod_{i=1}^{n} (y_i p(y_1 \ldots y_{i-1}) + (1 - y_i)(1 - p(y_1 \ldots y_{i-1}))).$$

Notice that according to this formula the probability that the ith bit is 1 is precisely $p(y_1 \ldots y_{i-1})$, a number between δ and $1-\delta$ *that depends in an arbitrary way on all previous outcomes.* In other words, bits of the sequence may bias the probabilities of subsequent bits in arbitrarily complex ways; but this bias may never make an outcome more certain than $1 - \delta < 1$. Thus, a $\frac{1}{2}$–random source is a perfect random source. A δ–random source with $\delta < \frac{1}{2}$ will be termed *slightly random.*

Example 11.2: Since a slightly random source is allowed to have strong dependences between bits, it is a much more realistic (and physically plausible and implementable) model than the perfect random source. Indeed, there is a host of physical processes that are arguably slightly random (Geiger counters, Zehner diodes, coins, capricious friends). Unfortunately, slightly random sources *appear to be useless for running randomized algorithms.*

Suppose that a Monte Carlo algorithm, say the random walk algorithm for 2SAT described in Section 11.1, is driven by the random bits generated by a δ–random source S_p where δ is much smaller than $\frac{1}{2}$. Depending on p, the probability of false negatives may become substantially larger than $\frac{1}{2}$.

For example, suppose that the instance of 2SAT has just one satisfying truth assignment $\hat{T}$, and define p in such a way that the source S_p biases our choices of literals to flip so that literals that agree with $\hat{T}$ are flipped with probability $1 - \delta$. It is not hard to see that the random walk algorithm, driven by the δ random source S_p with any $\delta < \frac{1}{2}$, needs exponential time to discover $\hat{T}$ with any decent probability.

Naturally, there are δ–random sources $S_{p'}$ that correctly drive the random walk algorithm —for a trivial example, define $p'(x) = \frac{1}{2}$ for all bit sequences x. But since we assume that we know nothing about p, we must be prepared for all possibilities. In other words, when a slightly random source S_p drives a randomized algorithm, we must assume that the values of p are set by an *adversary* who knows our algorithm, monitors its execution including its random choices, and maliciously strives to minimize its probability of success. In the case of the random walk algorithm for 2SAT the adversary can indeed reduce the probability of success to an insignificant trickle. □

So, slightly random sources cannot directly drive randomized algorithms. One may still hope that, through a sophisticated construction along the lines of the one explained in Example 11.1, a δ–random source could be used to generate truly random sequences. Unfortunately, it can be formally proven that no such construction is possible (see the references in 11.5.20).

Despite these setbacks, we shall next show that slightly random sources are very useful indeed: Although they cannot be used to directly drive randomized algorithms, or generate random bits, they can *simulate any randomized algorithm of interest with polynomial loss of efficiency*. To define exactly what this means, we must define formally randomized algorithms of the **RP** and **BPP** variety that use slightly random sources.

Definition 11.3: Let N be a precise, polynomially bounded nondeterministic Turing machine with exactly two choices per step, of the kind we used for defining **RP** and **BPP** (Definitions 11.1 and 11.2). The two choices available at each step are denoted the 0–*choice* and the 1–*choice*. On input x the computation $N(x)$ is in effect a *full binary tree* of depth $n = p(|x|)$ (see Figure 11.6; notice that throughout this section n denotes the length of the computation, not of the input). This tree has $2^{n+1} - 1$ nodes of which 2^n are leaves ("yes" or "no" answers) and $2^n - 1$ are internal. The tree has $2^{n+1} - 2$ edges, each corresponding to one of the two choices from an internal node (see Figure 11.6).

Let δ be a number between 0 and $\frac{1}{2}$. A *δ–assignment* F to $N(x)$ is a mapping from the set of edges of $N(x)$ to the interval $[\delta, 1 - \delta]$ such that the two edges leaving each internal node are assigned numbers adding up to one. For example, Figure 11.6 shows a .1 assignment to $N(x)$. Intuitively, a δ–assignment F captures the effect of the randomized algorithm N on input x driven by an arbitrary δ random source S_p. Function p is precisely the assignment F on the

1 choice out of an internal node, when each internal node is interpreted as the string of choices that leads to it.

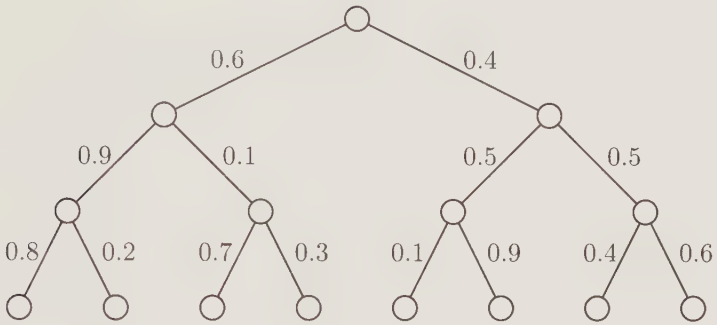


Figure 11-6. Computation tree and .1-assignment.

Given a δ–assignment F, for each leaf (final configuration) ℓ of $N(x)$ define the probability of ℓ to be $\mathbf{prob}[\ell] = \prod_{a \in P[\ell]} F(a)$, where $P[\ell]$ is the path from the root to leaf ℓ; that is, the probability that a leaf is reached is precisely the probability that all choices leading to that leaf are taken. Finally, define $\mathbf{prob}[M(x) = \text{"yes"} | F]$ to be the sum of $\mathbf{prob}[\ell]$ for all "yes" leaves ℓ of $N(x)$.

We are at last in a position to define the "slightly random" variants of the probabilistic classes **RP** and **BPP**. We say that a language L is in δ-**RP** if there is a nondeterministic Turing machine N, standardized as above, for which the following holds: If $x \in L$ then $\mathbf{prob}[M(x) = \text{"yes"} | F] \geq \frac{1}{2}$, and if $x \notin L$ then $\mathbf{prob}[M(x) = \text{"yes"} | F] = 0$ *for all* δ*–assignments* F. In other words, a language is in δ-**RP** if it has a randomized algorithm which, if run with any δ–random source, has no false positives, and has false negatives with probability less than half.

Similarly, a language L is in δ-**BPP** if there is a nondeterministic Turing machine N for which the following is true: If $x \in L$ then $\mathbf{prob}[M(x) = \text{"yes"} | F] \geq \frac{3}{4}$, and if $x \notin L$ then $\mathbf{prob}[M(x) = \text{"no"} | F] \geq \frac{3}{4}$, *for all* δ*–assignments* F. That is, the randomized algorithm is now required to decide correctly and by a clear margin when driven by any δ–random source. □

It is not hard to see that 0-**RP** = 0-**BPP** = **P**. The reason is that a 0–assignment may give probability one to any leaf of $N(x)$, and thus all leaves must agree on the outcome: The "randomized" algorithm must in fact be deterministic. It is also clear that $\frac{1}{2}$-**RP** = **RP** and $\frac{1}{2}$-**BPP** = **BPP**. But what about δ-**RP** and δ-**BPP** for intermediate values, $0 < \delta < \frac{1}{2}$? Is their power somehow intermediate between the corresponding classes and **P**? In other words, how much of the apparent power of randomization do we have to give up for the physical plausibility of slightly random sources?

The following important result states that no power is lost; randomization is practically implementable after all:

Theorem 11.4: For any $\delta > 0$, δ-**BPP** = **BPP**.

Proof: First, it is clear that δ-**BPP** $\subseteq$ **BPP**. So, assume that $L \in$ **BPP**; we shall show that $L \in \delta$-**BPP**. That is, from a machine N that decides L by clear majority we shall construct a machine N' that decides L also by clear majority, when driven by any δ–random source. We shall assume that the probability that N's answer is wrong is at most $\frac{1}{32}$, not the usual $\frac{1}{4}$; we saw in Section 11.2 that this is easy to achieve by repeating the algorithm enough times.

We shall describe machine N' as a randomized algorithm driven by a δ–source S_p. On input x, let $n = p(|x|)$ be the length of N's computation on x, and let $k = \lceil \frac{\log n + 5}{2\delta - 2\delta^2} \rceil$; k will be an important parameter in the simulation. A sequence of k bits will be called a *block*. Obviously there are 2^k possible blocks, denoted by the corresponding binary integers $0, 1, \ldots, 2^k - 1$. If $\kappa = (\kappa_1, \ldots, \kappa_k)$ and $\lambda = (\lambda_1, \ldots, \lambda_k)$ are blocks, then their *inner product* is defined $\kappa \cdot \lambda = \sum_{i=1}^{k} \kappa_i \lambda_i \bmod 2$. Notice that the inner product of two blocks is a bit.

Suppose that we have obtained a block $\beta_1 < 2^k$ (we denote blocks by Greek letters) by generating k consecutive random bits from our δ random source S_p. As we have seen, the bits in β_1 are not directly usable in simulating N, because our the hypothetical "adversary" may bias them in order to lead our algorithm to false positive or negative answers. Our strategy for "confusing" the adversary is simple: We use the k δ random bits in β_1 to generate in a perfectly deterministic way 2^k bits, namely $\beta_1 \cdot 0, \beta_1 \cdot 1, \ldots, \beta_1 \cdot (2^k - 1)$. (At this point it is a relief that k is only logarithmic in n, and so 2^k is polynomial.) *We then run 2^k simulations of N on input x in parallel*, where each of the 2^k bits is used as the first random bit required by N on input x. We repeat this n times (where n is the length of N's computation on x): At the jth repetition we generate a new block β_j, we "shatter it" into 2^k bits, and use them to advance the 2^k computations of N on x. In other words, we simulate N on input x for the following test sequences of choices: $T = \{(\beta_1 \cdot \kappa, \ldots, \beta_n \cdot \kappa) : \kappa = 0, \ldots, 2^k - 1\}$. Of the resulting 2^k answers ("yes" and "no") we adopt the majoritarian one as N''s answer on input x (if we have a tie, we answer "yes", say). This completes the description of N'.

Clearly N' is a randomized algorithm that works within time $\mathcal{O}(n2^k) = \mathcal{O}((p(|x|)^{1 + \frac{1}{2\delta - 2\delta^2}}))$, a polynomial in $|x|$. It remains to show that, no matter which δ random source S_p we use for generating the n blocks, the probability of a false answer is at most $\frac{1}{4}$.

Consider the set $\{0,1\}^n$ of possible sequences of choices for N on input x. Some of them will be *bad* false negatives or false positives, depending on whether x is or is not in L. We denote this set of bad sequences $B \subseteq \{0,1\}^n$. Since N is a **BPP** randomized algorithm, we know that $|B| \leq \frac{1}{32} 2^n$. The

probability of a false answer by N' is thus exactly $\mathbf{prob}[|T \cap B| \geq \frac{1}{2}|T|]$. To prove that N' decides L with clear majority when driven by any δ–random source we need to prove the following:

Claim: $\mathbf{prob}[|T \cap B| \geq \frac{1}{2}|T|] < \frac{1}{4}$.

The proof of the claim is rather indirect. We consider any one of the $n2^k$ bits used in our simulation, say $\beta_j \cdot \kappa$, where $1 \leq j \leq n$ and $0 \leq \kappa \leq 2^k - 1$. Define the *bias* of this bit to be, naturally enough, $(\mathbf{prob}[\beta_j \cdot \kappa = 1] - |\mathbf{prob}[\beta_j \cdot \kappa = 0])^2$. Clearly, by biasing the k bits in β_j, the adversary may bias a lot any single bit $\beta_j \cdot \kappa$; the hope is that most of these $2^k - 1$ bits will remain relatively unbiased. We would thus like to bound from above the *average bias* of a bit, averaged over all κ. It turns out that this can be done in two steps: We first show a totally unexpected connection between this average bias and *the coincidence probability of blocks*–that is, the probability that if the experiment of drawing a block from the source is repeated twice, the same block will result. In a second step, we bound this probability from above.

For the first step, define the coincidence probability to be $\sum_{\beta=0}^{2^k-1} p[\beta]^2$, where by $p[\beta]$ we denote the probability that our δ–random source will generate block β. The following result states that *the average bias equals the coincidence probability.*

Lemma 11.10: $\frac{1}{2^k} \sum_{\kappa=0}^{2^k-1} (\mathbf{prob}[\beta \cdot \kappa = 1] - |\mathbf{prob}[\beta \cdot \kappa = 0])^2 = \sum_{\beta=0}^{2^k-1} p[\beta]^2$.

Proof: This result has a clever algebraic proof. The key observation is that $\mathbf{prob}[\beta \cdot \kappa = 0] - |\mathbf{prob}[\beta \cdot \kappa = 1] = \sum_{\beta=0}^{2^k-1} (-1)^{\beta \cdot \kappa} p[\beta]$. This is true because the factor $(-1)^{\beta \cdot \kappa}$ is 1 whenever $\beta \cdot \kappa = 0$ and -1 when $\beta \cdot \kappa = 1$. Thus

$$\sum_{\kappa=0}^{2^k-1} (\mathbf{prob}[\beta \cdot \kappa = 1] - |\mathbf{prob}[\beta \cdot \kappa = 0])^2 =$$

$$\sum_{\kappa=0}^{2^k-1} \left(\sum_{\beta=0}^{2^k-1} (-1)^{\beta \cdot \kappa} p[\beta] \right)^2 =$$

$$\sum_{\kappa=0}^{2^k-1} \sum_{\beta=0}^{2^k-1} p[\beta]^2 + 2 \sum_{\kappa=0}^{2^k-1} \sum_{\beta,\beta'=0}^{2^k-1} (-1)^{(\beta+\beta') \cdot \kappa} p[\beta] p[\beta'].$$

However, if we reverse the order of summation in the second term we get $2 \sum_{\beta,\beta'=0}^{2^k-1} p[\beta] p[\beta'] (\sum_{\kappa=0}^{2^k-1} (-1)^{(\beta+\beta') \cdot \kappa})$, and it is easy to see that the inner sum is zero. □

The next lemma says that the coincidence probability of a block is at most equal the kth power of the coincidence probability of each bit:

Lemma 11.11: If β is a block produced by generating k bits of a δ-random source, then $\sum_{\beta=0}^{2^k-1} p[\beta]^2 \leq (\delta^2 + (1-\delta)^2)^k$.

Proof: Let $\beta = (x_1, \ldots, x_k)$, and suppose that the probability that $x_i = 1$ (as set by the δ-random source) is p_i. Consider now a block β' that differs from β only in that in β we have $x_i = 1$ and in β' $x_i = 0$. It is clear that the expressions for $p[\beta]$ and $p[\beta']$ are of the form Ap_i and $A(1-p_i)$. Separating the summation $\sum_{\beta=0}^{2^k-1} p[\beta]^2$ into those β's for which $x_i = 1$ and those for which $x_i = 0$, we conclude that the sum is of the form $Bp_i^2 + B(1-p_i)^2$ for some $B > 0$. It follows that the sum gets its maximum value if p_i and $1 - p_i$ differ as much as possible, that is, if one of them is δ and the other $1 - \delta$. This holds for all k bits. Thus, the maximum value of the sum is

$$\sum_{i=0}^{k} \binom{k}{i} \delta^{2i}(1-\delta)^{2(k-i)} = (\delta^2 + (1-\delta)^2)^k.$$

completing the proof of the lemma. □

By the two lemmata above, the total bias of the bits at the jth stage is no more than $2^k(\delta^2 + (1-\delta)^2)^k$. Let us call a bit $\beta_j \cdot \kappa$ *unbiased* if its bias $(\mathbf{prob}[\beta_j \cdot \kappa = 1] - \mathbf{prob}[\beta_j \cdot \kappa = 0])^2$ is at most $\frac{1}{n^2}$; otherwise the bit is *biased*. Notice that, if a bit is unbiased, its probability of being one is between $\frac{1}{2} - \frac{1}{2n}$ and $\frac{1}{2} + \frac{1}{2n}$.

It follows from the two lemmata that *there are at most* $n^2 2^k(\delta^2 + (1-\delta)^2)^k$ *biased bits* at the jth stage, and thus no more than $n^3 2^k(\delta^2 + (1-\delta)^2)^k$ biased bits overall. This expression can be bounded from above by recalling the value of k:

$$\begin{aligned} 2^k n^3(\delta^2 + (1-\delta)^2)^k = \\ 2^k n^3(1 - 2\delta + 2\delta^2)^{\frac{3\log n + 5}{2\delta - 2\delta^2}} = \\ 2^k n^3 2^{\log(1-2\delta+2\delta^2)\frac{3\log n+5}{2\delta-2\delta^2}} \leq \\ 2^k n^3 2^{-3\log n - 5} = \frac{1}{32} 2^k. \end{aligned}$$

To get the last line we recalled that $\log(1-\epsilon) \leq -\epsilon$ for all $0 < \epsilon < 1$. We conclude that there are at most $\frac{1}{32}2^k$ biased bits.

Now recall that N' works by simulating N on each of the 2^k sequences in $T = \{(\beta_1 \cdot \kappa, \ldots, \beta_n \cdot \kappa) : \kappa = 0, \ldots, 2^k\}$. Call a sequence in T *biased* if it contains at least one biased bit, and let $U \subseteq T$ denote the set of all unbiased sequences. By the previous paragraph there are at most $\frac{1}{32}2^k$ biased sequences (in the unlikely event that each biased bit makes a different sequence unbiased).

The proof of the claim (and the theorem) is now one calculation away: The expected value of $|B \cap T|$ (recall the claim being proved) is precisely

$$\mathcal{E}(|B \cap T|) = \sum_{t_1 \ldots t_n \in T} \sum_{b_1 \ldots b_n \in B} \prod_{i=1}^{n} \mathbf{prob}[b_i = t_i] \leq$$

$$\frac{1}{32}2^k + \sum_{t_1 \ldots t_n \in U} \sum_{b_1 \ldots b_n \in B} \prod_{i=1}^{n} \mathbf{prob}[b_i = t_i] \leq$$

$$\frac{1}{32}2^k + \sum_{t_1 \ldots t_n \in U} \sum_{b_1 \ldots b_n \in B} (\frac{1}{2} + \frac{1}{2n})^n \leq$$

$$\frac{1}{32}2^k + 2^k(\frac{1}{32}2^n)e2^{-n} <$$

$$\frac{1}{8}2^k = \frac{1}{8}|T|.$$

To obtain the second line we assume that, at worst, all biased sequences are bad. For the third, we know that the t_i's are unbiased and hence $\mathbf{prob}[t_i = b_i] \leq \frac{1}{2} + \frac{1}{2n}$. For the fourth line we recall the size of T, the upper bound on $|B|$, as well as that $(1 + \frac{1}{n})^n < e$.

Thus, we have that $\mathcal{E}(|B \cap T|) \leq \frac{1}{8}|T|$. By Lemma 11.2 (with $k = 4$) we have the claim, completing the proof that δ-**BPP** = **BPP**. □

Corollary: For any $\delta > 0$, δ-**RP** = **RP**.

Proof: The same algorithm that we used to simulate a **BPP** algorithm also simulates any **RP** algorithm, in such a way that the probability of false negatives is bounded. □

11.4 CIRCUIT COMPLEXITY

This is a good time for introducing an interesting point of view of complexity, based on *Boolean circuits*. We know from Chapter 4 that a Boolean circuit with n variable inputs, where n is a fixed number, can compute any Boolean function of n variables. Equivalently, we can think that a circuit accepts certain strings of length n in $\{0,1\}^*$, and rejects the rest. Here a string $x = x_1 \ldots x_n \in \{0,1\}^n$ is interpreted as a truth assignment to the input variables of the circuit, where the truth value of the ith input is **true** if and only if the symbol x_i is 1. However, this correspondence is good only for strings of a fixed length n. In order to relate circuits with arbitrary languages over the alphabet $\{0,1\}$, we need one circuit for each possible length of the input string.

Definition 11.4: The *size* of a circuit is the number of gates in it. A *family of circuits* is an infinite sequence $\mathcal{C} = (C_0, C_1, \ldots)$ of Boolean circuits, where C_n has n input variables. We say that a language $L \subseteq \{0,1\}^*$ *has polynomial*

circuits if there is a family of circuits $\mathcal{C} = (C_0, C_1, \ldots)$ such that the following are true: First, the size of C_n is at most $p(n)$ for some fixed polynomial p. And second, for all $x \in \{0,1\}^*$, $x \in L$ if and only if the output of $C_{|x|}$ is **true**, when the ith input variable is **true** if $x_i = 1$, and **false** otherwise. □

Example 11.3: What kinds of languages have polynomial circuits? We have already seen what essentially is a polynomial family of circuits for REACHABILITY, in Example 8.2. The inputs of the circuit are the entries of the adjacency matrix, and the Boolean circuit essentially computes the transitive closure of the graph. There is a different circuit for each number m of nodes in the graph. The number of inputs of this circuit is $n = m^2$; we can assume that the family is completed, for each k that is not a perfect square, by a Boolean circuit with k inputs, a **false** output, and with no other gates (so that no string which is not an adjacency matrix can be accepted).

The output of the circuit is the $(1, m)$ entry of the transitive closure (where as usual node 1 is the source and m is the destination). The size of the circuit for m nodes is exactly $\Theta(m^3)$. □

It is not totally accidental that REACHABILITY has polynomial circuits:

Proposition 11.1: All languages in **P** have polynomial circuits.

Proof: The construction in the proof of Theorem 8.1 gives, for each language $L \in \mathbf{P}$, decided by a Turing machine in time $p(n)$, and for each input x, a variable-free circuit with $\mathcal{O}(p(|x|)^2)$ gates (where the constant depends only on L) such that the output is **true** if and only if $x \in L$. It is easy to see that, when $L \subseteq \{0,1\}^*$, we can easily modify the input gates of the circuit so that they are variables reflecting the symbols of x. □

How about the converse of Proposition 11.1? Are all languages with polynomial circuits in **P**? The converse fails in the most dramatic way possible:

Proposition 11.2: There are undecidable languages that have polynomial circuits.

Proof: Let $L \subseteq \{0,1\}^*$ be *any* undecidable language in the alphabet $\{0,1\}$, and let $U \subseteq \{1\}^*$ be the language $U = \{1^n : \text{the binary expansion of } n \text{ is in } L\}$. Thus, U is a *unary language* (over a single-symbol alphabet). It is clear that U is undecidable, because the undecidable language L reduces to it (admittedly, by an exponential-time reduction, but such distinctions are insignificant in the context of undecidability).

Still, U has a trivial family of polynomial circuits $(C_0, C_1, \ldots)$. If the $1^n \in U$, then C_n consists of $n-1$ AND gates that take the conjunction of all inputs. Thus, the output is **true** if and only if the input is 1^n, as it ought to. If $1^n \notin U$, then C_n consists of its input gates, plus an output gate that is **false** (it has no edges). Thus, for all inputs C_n outputs **false**, as it should since there is no string of length n in U. □

Proposition 11.2 reveals a flaw in families of circuits as a realistic model of computation: We are allowed to invest unbounded amounts of computation in order to construct each circuit in the family (in the construction of Proposition 11.2, we had to solve an unsolvable problem...). This is clearly unacceptable, and leads us to the following definition:

Definition 11.4 (continued): A family $\mathcal{C} = (C_0, C_1, \ldots)$ of circuits is said to be *uniform* if there is a $\log n$-space bounded Turing machine N which on input 1^n outputs C_n. We say that a language L has *uniformly polynomial circuits* if there is a uniform family of polynomial circuits $(C_0, C_1, \ldots)$ that decides L. □

Example 11.3 (continued): It should be clear that the family of Boolean circuits for REACHABILITY are uniform. For any n we can construct in $\log n$ space the appropriate circuit. In contrast, the circuit family described in the proof of Proposition 11.6 is evidently *not* a uniformly polynomial one. □

In fact, uniformity is precisely the needed condition that identifies polynomial circuits with polynomial computation:

Theorem 11.5: A language L has uniformly polynomial circuits if and only if $L \in \mathbf{P}$.

Proof: One direction has already been proved: The construction of C_n in the proof of Theorem 8.1 can be done in $\mathcal{O}(\log n)$ space.

For the other direction, suppose that L has a uniformly polynomial family of circuits. Then we can decide whether an input x is in L by building $C_{|x|}$ in $\log |x|$ space (and hence in polynomial time), and then evaluating it in polynomial time, with the inputs set so that they spell x. □

Still, the precise power of "non-uniformly" polynomial circuits is of great interest. One reason is its possible relation with the $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ problem. Indeed, in view of Theorem 11.5, the $\mathbf{P} \neq \mathbf{NP}$ conjecture is equivalent to the following:

Conjecture A: **NP**-complete problems have no uniformly polynomial circuits.

The following stronger conjecture has been most influential:

Conjecture B: **NP**-complete problems have no polynomial circuits, *uniform or not.*

This hypothesis is not at all far-fetched. In Section 17.2 we prove a result that significantly supports it. Also, we already know that small circuit size is a rarity among Boolean functions (recall Theorem 4.3 and Problem 4.4.14). Thus, much of the effort in proving $\mathbf{P} \neq \mathbf{NP}$ in recent years has been directed towards proving Conjecture B, that is, showing that some specific **NP**-complete problem has no polynomial circuits (see Section 14.4 for a most interesting first step in this direction).

The following result suggests that circuits are useless in proving $\mathbf{P} \neq \mathbf{BPP}$:

Theorem 11.6: All languages in **BPP** have polynomial circuits.

Proof: Let $L \in \mathbf{BPP}$ be a language decided by a nondeterministic machine N that decides by clear majority. We claim that L has a polynomial family of circuits $(C_0, C_1, \ldots)$.

For each n, we shall describe now how to construct C_n. Obviously, if this description were explicit and simple, then by Theorem 11.5 we would have proved something remarkable: That $\mathbf{P} = \mathbf{BPP}$. Hence, our proof of existence of C_n will contain a step that is not "efficiently constructive." There is a very useful and elegant methodology, "the probabilistic method in combinatorics," (see the references) that yields such proofs. The current proof is a simple application of this technique; we shall see more complex examples later in this book.

Our circuit C_n is based on a sequence of bit strings $A_n = (a_1, \ldots, a_m)$ with $a_i \in \{0,1\}^{p(n)}$ for $i = 1, \ldots, m$, where $p(n)$ is the length of the computations of N on input of length n, and $m = 12(n+1)$. Each bit string $a_i \in A_n$ represents a possible sequence of choices for N, and so it completely specifies the computation of N on an input of length n. Informally, C_n, on input x, simulates N with each sequence of choices in A_n, and then takes the majority of the outcomes m. Since we know how to simulate polynomial computations by circuits, it is clear that, given A_n, we can construct C_n so that it has polynomially many gates.

But we must argue that there is an A_n such that C_n works correctly. That is, we must show the following result. (As usual we call a bit string bad if it leads N to either a false positive or a false negative answer.)

> **Claim:** For all $n > 0$ there is a set A_n of $m = 12(n+1)$ bit strings such that for all inputs x with $|x| = n$ fewer than half of the choices in A_n are bad.
>
> ***Proof:*** Consider a sequence A_n of m bit strings of length $p(n)$ selected *at random* by m independent samplings of $\{0,1\}^{p(n)}$. We ask the following question: *What is the probability that for each $x \in \{0,1\}^n$ more than half of the choices in A_n are correct?* We shall show that this probability is at least $\frac{1}{2}$.
>
> For each $x \in \{0,1\}^n$ at most one quarter of the computations are bad. Since the sequences in A_n were picked randomly and independently, the expected number of bad ones is $\frac{1}{4}m$. By the Chernoff bound (Lemma 11.9), the probability that the number of bad bit strings is $\frac{1}{2}m$ or more is at most $e^{-\frac{m}{12}} < \frac{1}{2^{n+1}}$.
>
> Now this last inequality holds for each $x \in \{0,1\}^n$. Thus, the probability that there is an x with no accepting sequence in A_n is at most the sum of these probabilities among all $x \in \{0,1\}^n$; and this sum is at most $2^n \frac{1}{2^{n+1}} = \frac{1}{2}$. We must conclude that, with probability at least half, our random selection of sequences has the desired property.

To put it another way, consider the space of all $2^{p(n)12(n+1)}$ possible selections of $12(n+1)$ bit strings (Figure 11.7). A small subset S_x of these, of cardinality at most $\frac{2^{p(n)12(n+1)}}{2^{n+1}}$, fails to provide the right answer by majority for input x. Certainly, the union of all these S_x's cannot have more than $2^n \frac{2^{p(n)12(n+1)}}{2^{n+1}}$ elements. Subtracting from $2^{p(n)12(n+1)}$, we conclude that at least half of the elements of the space are selections that have an accepting choice for each x.

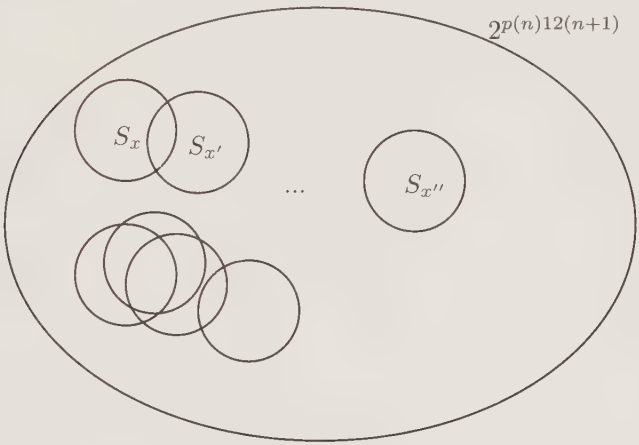


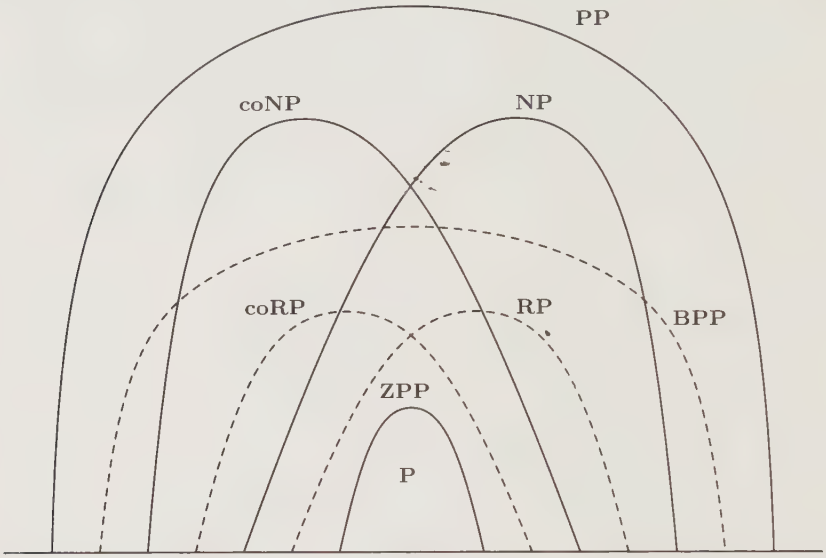
Figure 11-7. Counting sequences.

Notice that, although we are convinced that such an A_n must exist, as promised we have no idea how to find it... □

The proof of the theorem is now complete: Given such an A_n we can build a circuit C_n with $\mathcal{O}(n^2p^2(n))$ gates that simulates N with each of these sequences of choices, and then takes the majority of the outcomes. It follows from the property of A_n that C_n outputs **true** if and only if the input is in $L \cap \{0,1\}^n$. Hence, L has a polynomial family of circuits. □

11.5 NOTES, REFERENCES, AND PROBLEMS

11.5.1 Class review (semantic classes are shown in broken lines):



11.5.2 The slightly outrageous statement in the chapter header assumes that there is at least one meteorite impact per millennium that devastates at least 100 square meters on the earth surface.

11.5.3 Problem: (a) Show that all entries of the intermediate matrices in Gaussian elimination are rational numbers, with numerators and denominators that are subdeterminants of the original matrix.

(b) Conclude that no number in the course of Gaussian elimination ever has more than n^3 bits, where n is the size of the input.

11.5.4 Problem: We are given a matrix with entries that are either 1, 0, or one of the variables $x_1, \ldots, x_n$. We are asked whether the determinant contains a nonzero multiple of the term $x_1 \cdot x_2 \cdot \ldots \cdot x_n$. Show that this problem is **NP**-complete. (See Section 17.1 for a connection between determinants and directed graphs. Reduce the HAMILTON PATH problem for directed graphs to this one.)

11.5.5 Problem: Show that the random walk algorithm applied to any satisfiable expression in conjunctive normal form will converge to a satisfying truth assignment after $\mathcal{O}(n^n)$ expected number of steps. (What is the probability that it will choose the right moves one after the other, and find the satisfying truth assignment immediately? How many tries do we need so that this very improbable event is likely to eventually happen?)

11.5.6 Problem: Consider a Boolean expression $(x_1) \wedge (x_2) \wedge \ldots \wedge (x_n)$; obviously

it has just one satisfying truth assignment, the all-**true** one. Add now all clauses $(x_i \vee \neg x_j \vee \neg x_k)$, for all distinct indices $i, j, k \leq n$. Show that the random walk algorithm (even with randomization in the starting solution and choice of clause) performs very badly on this satisfiable instance of 3SAT.

11.5.7 The randomized algorithm for symbolic determinants was pointed out in

- J. T. Schwartz "Fast probabilistic algorithms for verification of polynomial identities," *J.ACM, 27*, pp. 710–717, 1980, and
- R. E. Zippel "Probabilistic algorithms for sparse polynomials," *Proc. EUROSAM '79*, pp. 216–226, Lecture Notes in Computer Science 72, Springer-Verlag, Berlin, 1979.

The randomized algorithm for 2SAT is from

- C. H. Papadimitriou "On selecting a satisfying truth assignment," *Proc. 32nd IEEE Symp. on the Foundations of Computer Science*, pp. 163–169, 1991.

But it was the randomized tests for primality that stirred interest in randomized computation in the late 1970's; the primality test in the text is from

- R. Solovay and V. Strassen "A fast Monte-Carlo test for primality," *SIAM J. Comp., 6*, pp. 84–86, 1977.

Another primality test, due to Michael Rabin, is the subject of Problem 11.5.10

- M. O. Rabin "Probabilistic algorithm for testing primality," *J. Number Theory, 12*, pp. 128–138, 1980.

It turns out that the problem of recognizing primes is not only in **coRP** as established by these results, *but it is in fact in* **RP**, *and thus in* **ZPP**:

- L. Adleman and M. Huang "Recognizing primes in random polynomial time," *Proceedings of the 19th ACM Symp. on the Theory of Computing*, pp. 462–470, 1987.

Furthermore, there is a *deterministic* polynomial-time algorithm for deciding whether a number is prime *if one assumes the Riemann hypothesis*, a most important number-theoretic conjecture concerning the roots of the Riemann ζ function and the distribution of primes (see the books on number theory cited in the previous chapter):

- G. L. Miller "Riemann's hypothesis and tests for primality," *J.CSS, 13*, pp. 300–317, 1976.

This result is also discussed in Problem 11.5.10.

11.5.8 Problem: (a) Show that Euclid's algorithm for computing the greatest common divisor of two integers $x > y$ (Problem 10.4.7 and Lemma 11.7) can be extended to compute two integers A, B, possibly negative, such that $A \cdot x + B \cdot y = \gcd(x, y)$. (Suppose you have such numbers for x and $y \bmod x$, call them A' and B', and you also know $\lfloor \frac{x}{y} \rfloor$. What is the formula for A and B?)

(b) Based on (a) show that, given n and $m \in \Phi(n)$, one can compute the *inverse* of $m \bmod n$ (the unique integer m^{-1} such that $m \cdot m^{-1} = 1 \bmod n$), in $\mathcal{O}(\log^3 n)$ steps.

11.5.9 Problem: Show that for all primes p, $(2|p) = (-1)^{\frac{p^2-1}{8}}$. Generalize to $(2|M)$, for all odd integers M.

11.5.10 Problem: Recall the Fermat test of Figure 11.3, checking whether $a^{n-1} \neq 1 \bmod n$; if a passes the test it is called a *Fermat witness.*

Suppose now that $n-1 = 2^k m$, where m is odd, and suppose that for some $a < n$ we have $a^m \neq \pm 1 \bmod n$, and squaring this number $k-1$ times we get the integers $a^{m2^i}, i = 1, \ldots, k-1$, *none of them congruent to* $-1 \bmod n$. Then a will be called a *Riemann witness.*

(a) Show that if n has a Riemann witness then it is composite.

Gary Miller in his paper cited above proves that, if the Riemann Hypothesis is true, then there is a witness (Fermat *or* Riemann) with $\mathcal{O}(\log \log n)$ bits.

(b) Show that this implies that PRIMES is in **P** (assuming the Riemann Hypothesis is true).

(c) Show that, if n is composite, then there are at least $\frac{1}{2}n$ witnesses (Fermat or Riemann). Based on this, describe another polynomial Monte Carlo test of compositeness. (The fraction can be improved to $\frac{3}{4}n$, which is the best possible, see Michael Rabin's paper cited above.)

11.5.11 The formal study of randomized complexity classes began with

- J. Gill "Computational complexity of probabilistic Turing machines," *SIAM Journal on Computing, 6,* pp. 675–695, 1977, where most of the classes discussed here were defined and Theorem 11.2, among others, proved.

11.5.12 Problem: Show that the following problems are undecidable, given a precise nondeterministic machine M.

(a) Is it true that for all inputs either all computations reject, or at least half of them accept?
(b) Is it true that for all inputs either at least $\frac{3}{4}$ of the computations reject, or at least $\frac{3}{4}$ of them accept?
(c) Is it true that for all inputs at least one computation accepts?
(d) Given *two* such machines, is it true that for all inputs either the first has an accepting computation, or the second does, but never both?

(Notice that these results refer to the "semantic" complexity classes **RP**, **BPP**, **TFNP**, and **NP** $\cap$ **coNP**, respectively.)

11.5.13 Problem: Show that **RP**, **BPP**, and **PP** are closed under reductions.

11.5.14 Problem: Show that **BPP** and **RP** are closed under union and intersection.

11.5.15 Problem: Show that **PP** is closed under complement and symmetric difference.

11.5.16 Problem: (a) Show that MAJSAT is **PP**-complete.

(b) Show that THRESHOLD SAT is **PP**-complete, where the problem is defined as follows: "Given an expression ϕ and an integer K, is the number of satisfying truth assignments of ϕ at least equal to K?"

11.5.17 Problem: Let $0 < \epsilon < 1$ be a rational number. We say that $L \in \mathbf{PP}_\epsilon$ if there is a nondeterministic Turing machine M such that $x \in L$ if and only if at least an ϵ fraction of the computations are accepting. Show that $\mathbf{PP}_\epsilon = \mathbf{PP}$.

11.5.18 Problem: Show that, if $\mathbf{NP} \subseteq \mathbf{BPP}$, then $\mathbf{RP} = \mathbf{NP}$. (That is, if SAT can be solved by randomized machines, then it can be solved by randomized machines with no false positives, presumably by computing a satisfying truth assignment as in Example 10.3.)

11.5.19 For an interesting treatment of randomized complexity classes in terms of generalized quantifiers and their algebraic properties see

- S. Zachos "Probabilistic quantifiers and games," *J.CSS 36*, pp. 433–451, 1983.

11.5.20 Random sources. Von Neumann's technique for creating an unbiased source from a biased one (Example 11.1) is from

- J. von Neumann "Various techniques for use in connection with random digits," in *von Neumann's Collected Works*, pp. 768–770, Pergamon, New York, 1963.

For removing more complex bias (in the form of a known Markov chain) see

- M. Blum "Independent unbiased coin flips from a correlated biased source," *Proc. 25th IEEE Symp. on the Foundations of Computer Science*, pp. 425–433, 1983.

Slightly random sources were introduced in

- M. Santha and U. V. Vazirani "Generating quasi-random sequences from slightly random sources," *Proc. 25th IEEE Symp. on the Foundations of Computer Science*, pp. 434–440, 1984,

where it was proved that there is *no way* to generate random bits from such a source. However, if we have *two independent* slightly random sources, it can be shown that perfectly random bits can be generated:

- U. V. Vazirani "Towards a strong communication complexity, or generating quasi-random sequences from two communicating slightly random sources," *Proc. 17 ACM Symp. on the Theory of Computing*, pp. 366–378, 1985.

But of course, independence can be as difficult to find in nature as randomness. Theorem 11.4 (actually, its **RP** version, and the basic proof technique) is from

- U. V. Vazirani and V. V. Vazirani "Random polynomial time equals semi-random polynomial time," *Proc. 26th IEEE Symp. on the Foundations of Computer Science*, pp. 417–428, 1985.

Lemma 11.10 is a $0-1$ version of Parseval's theorem, see

- P. Halmos *Finite Dimensional Vector Spaces*, Springer Verlag, Berlin, 1967.

For an improvement on Theorem 11.4 see

- D. Zuckerman "Simulation of BPP using a general weak random source," *Proc. 32nd IEEE Symp. on the Foundations of Computer Science*, pp. 79–89, 1991.

The random sources considered in this paper are restricted so that values of *whole blocks* (instead of single bits) do not have much higher probability than their allotted

one.

11.5.21 That the commonly used congruential pseudorandom number generators are terrible (in the sense that it is easy to predict bits, even to deduce the "secret" parameters) is now well-known; see

- J. Plumstead, "Inferring a sequence generated by a linear congruence," in *Proc. 23rd IEEE Symp. on the Foundations of Computer Science*, pp. 153–159, 1983; and
- A. M. Frieze, J. Håstad, R. Kannan, J. C. Lagarias, and A. Shamir "Reconstructing truncated integer variables satisfying linear congruences," *SIAM J. Comp.*, *17*, 2, pp. 262–280, 1988.

"Provably" pseudorandom sequences can be generated using cryptographic techniques and making complexity assumptions stronger than $\mathbf{P} \neq \mathbf{NP}$; see 12.3.5 in the notes and references of the next chapter.

11.5.22 Problem: Show that if $L \in \mathbf{TIME}(f(n))$ for proper $f(n)$, then there is a uniform circuit family deciding L such that the nth circuit has $\mathcal{O}(f(n) \log f(n))$ gates. (This improves substantially on the $\mathcal{O}(f^4(n))$ bound implicit in the proof of Theorem 11.5. It can be achieved by employing an *oblivious* machine that simulates the original Turing machine in $\mathcal{O}(f(n) \log f(n))$ time, recall Problem 2.8.10. For oblivious machines one need have just one copy of the circuit C in the proof of Theorem 8.1 per step, since we know where the cursor is, and thus where changes are going to occur. This argument is from

- N. Pippenger and M. J. Fischer "Relations among complexity measures," *J.ACM*, *26*, pp. 423–432, 1979.)

11.5.23 The probabilistic method for SAT. (a) Suppose that a Boolean expression has fewer than n^k clauses, each with at least $k \log n$ distinct variables. Use the probabilistic method of the proof of Theorem 11.6 to show that it has a satisfying truth assignment.

(b) Give a polynomial-time algorithm that finds a satisfying truth assignment, given such an expression. (See the proof of Theorem 13.2 for a similar argument.)

11.5.24 Computation with advice. Suppose that our Turing machines have an extra read-only input string called the *advice string*, and let $A(n)$ be a function mapping integers to strings in Σ^*. We say that machine M decides language L *with advice* $A(n)$ if $x \in L$ implies $M(x, A(|x|)) =$ "yes", and $x \notin L$ implies $M(x, A(|x|)) =$ "no". That is, the advice $A(n)$, specific to the length of the input, helps M decide all strings of length n correctly. Let $f(n)$ be a function mapping nonnegative integers to nonnegative integers. We say that $L \in \mathbf{P}/f(n)$ if there is an advice function $A(n)$, where $|A(n)| \leq f(n)$ for all $n \geq 0$, and a polynomial-time Turing machine M with advice, such that M decides L with advice A.

This elegant way of quantifying nonuniformity was proposed in

- R. M. Karp and R. J. Lipton "Some connections between nonuniform and uniform complexity classes," *Proc. 12th ACM Symp. on the Theory of Computing*,

pp. 302–309, 1980.

(a) Prove that $L \in \mathbf{P}/n^k$ if and only if L has polynomial circuits.

(b) Prove that, if SAT$\in \mathbf{P}/\log n$, then $\mathbf{P} = \mathbf{NP}$. (Run through all possible advice strings, and find the correct one using self-reducibility; see the proof of Theorem 14.3 for a similar technique.)

(c) Define nondeterministic Turing machines with advice. Prove that $L \in \mathbf{NP}/n^k$ if and only if there is a family $\mathcal{C} = (C_1, C_2, \ldots)$ of circuits, where C_i has i inputs and i outputs, such that $L \cup 0^* = \{C_i(x) : i \geq 1, x \in \{0,1\}^i\}$.

(d) Prove that there are undecidable problems even in $\mathbf{P}/\log n$.

11.5.25 Problem: We know that most languages do not have polynomial circuits (Theorem 4.3), but that certain undecidable ones do (Proposition 11.2). We suspect that **NP**-complete languages have no polynomial circuits (Conjecture B in Section 11.4). How high do we have to go in complexity to find languages that *provably* do not have polynomial circuits?

Show that there is a language in *exponential space* that has no polynomial circuits. (In exponential space we can diagonalize over all possible polynomial circuits.)

Can the exponential space bound above be improved? It can be brought down to some level of the *exponential hierarchy* (see Chapters 17 and 20). Furthermore, unless **PSPACE** contains languages with no polynomial circuits, some very counterinituitive inclusions between complexity classes must hold, see the paper by Karp and Lipton cited above.

11.5.26 We shall study other aspects of circuit complexity in Chapters 14 and 17; for much more on this important subject see

- J. E. Savage *The Complexity of Computing,* Wiley, New York, 1976,
- I. Wegener *The Complexity of Boolean Functions,* Teubner, Stuttgard, 1987, and the survey
- R. B. Boppana and M. Sipser "The complexity of finite functions," pp. 757–804 in *The Handbook of Theoretical Computer Science, vol. I: Algorithms and Complexity*, edited by J. van Leeuwen, MIT Press, Cambridge, Massachusetts, 1990.

11.5.27 The distribution of primes. How many primes are there? We know that there are infinitely many of them, but how *densely* are the prime numbers distributed among the composites? Intuitively, the density of primes must be decreasing as the numbers get bigger, but what is the precise law?

The ultimate answer is given by the *prime number theorem*, a most important and deep result in number theory, see chapter XXII of

- G. H. Hardy and E. M. Wright *An Introduction to the Theory of Numbers,* Oxford Univ. Press, Oxford, U.K., 5th edition, 1979.

This theorem says that the number of primes up to n, denoted $\pi(n)$, is asymptotocally $\frac{n}{\ln n}$ *and the constant is one.* It implies that the "density" of primes around n is asymptotically $\frac{1}{\ln n}$.

Tchebychef's theorem provides rigorous and asymptotically tight bounds for $\pi(x)$, the number of primes less than x (although it does not provide the exact constant). Its proof goes as follows: Consider

$$N = \binom{2n}{n} = \frac{(n+1)(n+2)\cdots(2n)}{n!}; \tag{1}$$

it is an integer between 2^n and 2^{2n} (in fact, by Stirling's approximation N is about $\frac{2^{2n}}{\sqrt{2n}}$). All primes between n and $2n$ divide the numerator but not the denominator of the fraction in (1).

(a) Based on this, prove that $\pi(x) \leq \frac{2x}{\log x}$.

(b) Prove that $\pi(x) \geq \frac{x}{4 \log x}$ (establish first that

$$\log N \leq \sum_{p \leq 2n} \lfloor \frac{\log 2n}{\log p} \rfloor \log p).$$

Here is an informal argument that gives the correct answer: Let $f(x)$ be a function intuitively standing for the "density of primes near $x > 0$." Let us try to determine how $f(x)$ might change as x grows. Basically, $f(x)$ is the percentage of numbers that resist division by all primes $\leq \sqrt{x}$. Thus, $f(x + \Delta x)$, where $\Delta x > 0$ is a "small" increment, is going to be smaller than $f(x)$, because there are more primes $\leq \sqrt{x + \Delta x}$ than there are $\leq \sqrt{x}$. How many more? The answer is $(\sqrt{x + \Delta x} - \sqrt{x}) f(\sqrt{x})$, or about $\frac{\Delta x}{2} f(\sqrt{x})$, since Δx is assumed to be small. Now, these extra primes divide certain numbers around $x + \Delta x$ that were not divided by lesser primes, and therefore decrease $f(x)$. Each such prime divides about one in every $\sqrt{x}$ of these numbers, and thus decreases $f(x)$ by $\frac{f(x)}{\sqrt{x}}$. This accounts for the whole decrease of $f(x)$ between x and $x + \Delta x$, and thus we can write $f(x + \Delta x) - f(x) = -\frac{\Delta x}{2\sqrt{x}} f(x) f(\sqrt{x})$, or

$$\frac{df}{dx} = -\frac{f(x) f(\sqrt{x})}{2\sqrt{x}}. \tag{2}$$

(c) Prove that $\frac{1}{\ln x}$ obeys differential equation (2); in fact, it is *the only analytical function* that does.

CHAPTER

12 CRYPTOGRAPHY

Complexity is not always a disease to be diagnosed; sometimes it is a resource to be exploited. But complexity turns out to be most elusive precisely where it would be most welcome.

12.1 ONE-WAY FUNCTIONS

Cryptography deals with the following situation. Two parties (succumbing to a cute tradition we shall call them *Alice and Bob*) wish to communicate in the presence of malevolent eavesdroppers. That is, Alice wants to send a message to Bob, over a channel monitored by an adversary (Figure 12.1), and wishes the message to be known only to her and Bob.

Alice and Bob handle this situation as follows: They agree on two algorithms E and D—the *encoding* and the *decoding* algorithms. These algorithms are assumed to be known to the general public. Alice runs E, and wishes to send a message $x \in \Sigma^*$ (where $\Sigma = \{0, 1\}$ throughout this chapter) to Bob, who operates D. Privacy is assured in terms of two strings $e, d \in \Sigma^*$, the *encoding and decoding key*, respectively, known only to the communicating parties. Alice computes the *encrypted message* $y = E(e, x)$ and transmits it to Bob over the unreliable channel. Bob receives y, and computes $D(d, y) = x$. In other words, e and d are carefully selected so that they make D an inverse of E. Naturally, E and D should be polynomial-time algorithms, but there should be no way for an eavesdropper to compute x from y, without knowing d.

There is nothing deep or mysterious about how to achieve this. One can choose both d and e to be the same arbitrary string e of length $|x|$, and let both $E(e, x)$ and $D(e, y)$ to be simply the *exclusive or* of the corresponding strings

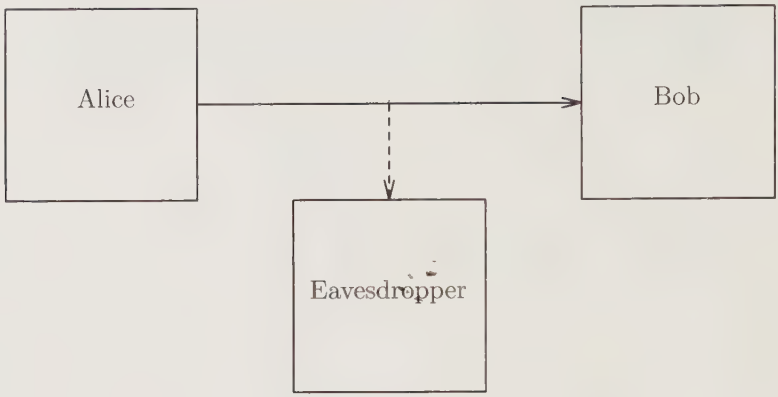


Figure 12-1. Alice, Bob, and friends.

$E(e,x) = e \oplus x$ and $D(e,y) = e \oplus y$. That is, the ith bit of $D(e,y)$ is one if and only if exactly one of e_i, y_i is one. This classical scheme, known as the *one-time pad*, obviously works as intended. First, since $((x \oplus e) \oplus e) = x$, we have that $D(d, E(e,x)) = x$, and the two functions are indeed inverses of one another, as required. Second, if an eavesdropper could derive x from y, then clearly he or she knows $e = x \oplus y$. Notice that this is a formal impossibility proof: No eavesdropper can deduce x from y without knowing e.

But there are problems with this scheme. First, the key must somehow be agreed upon, and this communication must obviously also be protected. Also, the keys must be as long as the message, and this makes frequent routine communication impossible by this method.

Public-Key Cryptography

Modern cryptography is based on an ingenious twist of this situation. Suppose that only d is secret and private to Bob, *while e is well-known to Alice and the general public.* That is, Bob generates the (e,d) pair and announces e openly. Alice (or anybody else for that matter) can send a message x to Bob by computing and transmitting $E(e,x)$, where as always $D(d, E(e,x)) = x$. The point is that *it is computationally infeasible to deduce d from e, and x from y without knowing d.* This setup is called a *public-key cryptosystem.*

In this situation we cannot hope for an impossibility proof like the one for the one-time pad. This is because the difficulty of compromising a public-key cryptosystem rests with the difficulty of guessing x from y. Once we have correctly guessed x, we can check whether it is the original message simply by testing whether $E(e,x) = y$. And since x cannot be more than polynomially longer than y, *compromising a public-key cryptosystem is a problem in* **FNP**.

And for all we know all such problems can be solved in polynomial time...

So, secure public key cryptosystems can exist only if $\mathbf{P} \neq \mathbf{NP}$. But even if we assume that $\mathbf{P} \neq \mathbf{NP}$, the existence of a secure public-key cryptosystem is not immediate. What is needed is a very special inhabitant of $\mathbf{FNP} - \mathbf{FP}$ called a *one-way function*.

Definition 12.1: Let f be a function from strings to strings. We say that f is a *one-way function* if the following hold:

(i) f is one-to-one, and for all $x \in \Sigma^*$, $|x|^{\frac{1}{k}} \leq |f(x)| \leq |x|^k$ for some $k > 0$. That is, $f(x)$ is at most polynomially longer or shorter than x.

(ii) f is in **FP**, that is, it can be computed in polynomial time.

(iii) The most important requirement is that f^{-1}, the inverse of f, *is not in* **FP**. That is, there is no polynomial-time algorithm which, given y, either computes an x such that $f(x) = y$ or returns "no," if such an x does not exist. Notice that, since f is one-to-one, x can be uniquely recovered from $f(x)$ — for example, by trying all x's of appropriate length. The point is that there is no polynomial-time algorithm that achieves this.

Notice that function f^{-1}, which is not supposed to be in **FP** by (iii) of this definition, is definitely in **FNP**. This is because we can check any alleged x by computing $f(x)$ and verifying that it is the given value y. □

Example 12.1: As we shall see, even if $\mathbf{P} \neq \mathbf{NP}$ there is no guarantee that one-way functions indeed exist. However, there is one function that many people suspect is indeed a one-way function: *Integer multiplication*. By this we do not mean exactly $f(x, y) = x \cdot y$ where x and y are arbitrary integers—this function is not one-to-one, consider $3 \cdot 4 = 2 \cdot 6$. But suppose that $p < q$ are prime numbers, and $C(p)$, $C(q)$ appropriate "certificates" of their primality (recall the corollary to Theorem 10.1). Then the function $f_{\text{MULT}}(p, C(p), q, C(q)) = p \cdot q$ (where f_{MULT} returns, say, its input if $C(p)$ and $C(q)$ fail to be valid primality certificates) is indeed one-to-one, and polynomial-time computable. And we know of no polynomial algorithm which inverts f—that is, factors products of large primes. Although there are *subexponential* algorithms for factoring integers (see Problem 10.4.11), at present we know of no polynomial-time one, or even an algorithm that would consistently factor products of two primes with many hundreds of bits. □

Example 12.2: There is another suspect one-way function, *exponentiation modulo a prime*. f_{EXP} takes as arguments a prime p with its certificate $C(p)$, a primitive root r modulo p (recall that such a certified root is included in $C(p)$), and an integer $x < p$. f_{EXP} returns $f_{\text{EXP}}(p, C(p), r, x) = (p, C(p), r^x \bmod p)$. Inverting f_{EXP} is another well-known hard computational problem in number theory called the *discrete logarithm problem*, for which no polynomial-time algorithm is known. Indeed, computing x from $r^x \bmod p$ is the discrete equivalent of computing the base-r logarithm of a residue. □

Example 12.3: f_{MULT} and f_{EXP} cannot be used directly as the basis of a public-key cryptosystem, but a clever combination of the two can. Let p and q be two prime numbers, and let consider their product $p \cdot q$. The number of bits of pq is $n = \lceil \log pq \rceil$ (in the intended cryptographic applications, n will be in the hundreds). All numbers modulo pq will be thus considered as n-bit strings over $\{0,1\}$, and vice-versa. Suppose that d is a number that is relatively prime to $\phi(pq) = pq(1-\frac{1}{p})(1-\frac{1}{q}) = pq - p - q + 1$ —this is the Euler function on pq, recall Lemma 10.1. The *RSA function* (after the initials of the researchers who proposed it, Ron Rivest, Adi Shamir, and Len Adleman) is this:

$$f_{\text{RSA}}(x, e, p, C(p), q, C(q)) = (x^e \bmod pq, pq, e).$$

That is, f_{RSA} simply raises x to the eth power modulo pq, and also reveals the product pq and the exponent e (but of course not p or q). We assume again that the output is the input itself if the input is wrong —if the C's are not valid certificates, or if e is not relatively prime to $\phi(pq)$.

Is the RSA function a one-way function? We will shortly show that it is one-to-one, as required (this is why we insisted that e be relatively prime to $\phi(pq)$). It clearly can be computed in polynomial time, and so it satisfies property (i) —remember, we exponentiate an n-bit number by repeated squaring, in $\mathcal{O}(n^3)$ time, recall Section 10.2 and Problem 10.4.7. Although, naturally, we do not hope to prove any time soon that f_{RSA} satisfies property (iii) in the definition of one-way functions, inverting f_{RSA} does seem to be a highly non-trivial problem. As with factoring (to which it can be reduced, see below), *despite intensive effort for many years, no polynomial algorithm for inverting the RSA function has been announced.* □

What is more important, the RSA function can be the basis of a public-key cryptosystem, which we describe next. Bob knows p and q, and announces their product pq, as well as e, an integer prime to $\phi(pq)$. This is the public encryption key, available to anybody who may wish to communicate with Bob. Alice uses the public key to encrypt message x, an n bit integer, as follows:

$$y = x^e \bmod pq.$$

Bob knows, besides what Alice knows, an integer d, another residue modulo pq such that $e \cdot d = 1 + k\phi(pq)$ for some integer k. That is, d is the inverse of e in the ring modulo $\phi(pq)$. Since e is relatively prime to $\phi(pq)$, this number exists and can be found by Euclid's algorithm (recall Lemma 11.6 and Problem 11.5.8). In order to decrypt y, Bob simply raises it to the dth power:

$$y^d = x^{e \cdot d} = x^{1+k\phi(pq)} = x \bmod pq,$$

simply because $x^{\phi(pq)} = 1 \bmod pq$ by Fermat's theorem (corollary to Lemma 10.3). To summarize the RSA public-key cryptosystem, the encryption key is

(pq, e), the decryption key is (pq, d), and both algorithms involve just modular exponentiation. (Incidentally, the latter equation also shows that f_{RSA} is one-to-one: d-th roots are unique, as long as d is relatively prime to $\phi(pq)$.)

Any algorithm that factors integers can be used to invert the RSA function efficiently. Once we have factored pq and know p and q, we would first compute $\phi(pq) = pq - p - q + 1$, and from it and e we would recover d by Euclid's algorithm, and finally $x = y^d \bmod pq$. Thus, inverting f_{RSA} can be reduced to inverting f_{MULT}. However, there could conceivably be more direct ways for decrypting in the RSA public-key cryptosystem, without factoring pq. There are variants of the cryptosystem for which the reduction goes the other way: These variants are exactly as hard to "break" as f_{MULT} (see the references).

Cryptography and Complexity

At this point it may seem very tempting to try to link the existence of one-way functions (and therefore of secure public-key cryptosystems) with the fragment of complexity theory that we know well, and has been so helpful in identifying credibly hard functions -NP-completeness. Unfortunately, there are two problems: First, *it cannot be done*. Second, *it is not worth doing*. To understand the obstacles, it is helpful to introduce a complexity class closely related to one-way functions.

Definition 12.2: Call a nondeterministic Turing machine *unambiguous* if it has the following property: For any input x there is at most one accepting computation. **UP** is the class of languages accepted by unambiguous polynomial-time bounded nondeterministic Turing machines. □

It is obvious that $\mathbf{P} \subseteq \mathbf{UP} \subseteq \mathbf{NP}$. For the first inclusion, a deterministic machine can be thought of as a nondeterministic one, only with a single choice at each step; such a "nondeterministic" machine must necessarily be unambiguous. For the second inclusion, unambiguous machines are by definition a special class of nondeterministic ones. The following result establishes that **UP** is intimately related to one-way functions.

Theorem 12.1: $\mathbf{UP} = \mathbf{P}$ if and only if there are no one-way functions.

Proof: Suppose that there is a one-way function f. Define now the following language: $L_f = \{(x, y) : \text{there is a } z \text{ such that } f(z) = y, \text{ and } z \leq x\}$. In writing $z \leq x$, we assume that all strings in $\{0,1\}^*$ are ordered, first by length, and strings of the same length n are ordered lexicographically, viewed as n-bit integers. That is, $\epsilon < 0 < 1 < 00 < 01 < 10 < 11 < 000 < \ldots$.

We claim that $L_f \in \mathbf{UP} - \mathbf{P}$. It is easy to see that there is an unambiguous machine U that accepts L_f: U on input (x, y) nondeterministically guesses a string z of length at most $|y|^k$ (recall (ii) in the definition of a one-way function), and tests whether $y = f(z)$. If the answer is "yes" (since f is one-to-one this will happen at most once), it checks whether $z \leq x$, and if so it accepts. It should

be clear that this nondeterministic machine decides L_f, and is unambiguous. Hence $L_f \in \mathbf{UP}$.

We have now to show that $L_f \notin \mathbf{P}$. Suppose that there is a polynomial-time algorithm for L_f. Then we can invert the one-way function f by *binary search*: Given y, we ask whether $(1^{|y|^k}, y) \in L_f$, where k is the integer in part (ii) of Definition 12.1. If the answer is "no,", this means that there is no x such that $f(x) = y$ —if there were such an x, it would have to be lexicographically smaller than $1^{|y|^k}$, since $|y| \geq |x|^{\frac{1}{k}}$. If the answer is "yes," then we ask whether $(1^{|y|^k-1}, y) \in L_f$, and then $(1^{|y|^k-2}, y) \in L_f$, and so on, until for some query $(1^{\ell-1}, y) \in L_f$ we get the answer "no", and thus determine the actual length $\ell \leq |y|^k$ of x. We then determine one-by-one the bits of x, again by asking whether $(01^{\ell-1}, y) \in L_f$ and then, depending on whether the answer was "yes" or "no," asking $(001^{\ell-2}, y) \in L_f$ or $(101^{\ell-2}, y) \in L_f$, respectively; and so on. After a total of at most $2n^k$ applications of the polynomial algorithm for L_f, we have inverted f on y.

Conversely, suppose that there is a language $L \in \mathbf{UP} - \mathbf{P}$. Let U be the unambiguous nondeterministic Turing machine accepting L, and let x be an accepting computation of U on input y; we define $f_U(x) = 1y$, that is, the input of U for which x is an accepting computation prefixed by the "flag" 1 (whose meaning is that $f_U(x)$ is indeed a corresponding input, and the need for which will become clear soon). If x does not encode a computation of U, $f_U(x) = 0x$—the flag now is 0 to warn us that the argument of f_U is not a computation.

We claim that f_U is a one-way function. It is certainly a well-defined function in $\mathbf{FP}$, because y is a part of the representation of the computation x and can be essentially "read off" x. Second, the lengths of argument and result are polynomially related, as required, because U has polynomially long computations. The function is one-to-one, because, since the machine is unambiguous, and we use flags, $f(x) = f(x')$ means that $x = x'$. And if we could invert f_U in polynomial time, then we would be able to decide L in polynomial time as well: Inverting f_U on $1y$ tells us whether U accepts y or not. □

We fully expect that $\mathbf{P} \neq \mathbf{UP}$. $\mathbf{UP} = \mathbf{NP}$ is another very unlikely event. It would mean that SAT can be decided by an unambiguous machine, one that does not try all truth assignments and fail at some, but purposefully zeroes in the correct satisfying truth assignment. Thus, the correct complexity context for discussing cryptography and one-way functions is the $\mathbf{P} \stackrel{?}{=} \mathbf{UP}$ question, not the $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ one. It is in this sense that $\mathbf{NP}$-completeness is not useful in identifying one-way functions. But neither can we hope to base such an argument on $\mathbf{UP}$-completeness. This is because of the distinctly semantic flavor of the definition of $\mathbf{UP}$: For all inputs there are either zero or one accepting computations (recall the discussion of classes such as $\mathbf{RP}$ in Section 11.2, where

the number of accepting computations is either zero or more than half of the total). As a result, **UP** is not known—or believed—to have complete problems.

But even if we *could* relate a cryptosystem with an **NP**-complete (or **UP**-complete) problem, this would be of very limited significance and use. There is a more fundamental reason why the complexity concepts we have studied so far in this book are inadequate for approaching the issue of secure cryptography: We have based our study of complexity on the *worst-case* performance of algorithms. This is reasonable and well-motivated when studying computational problems, where we want algorithms whose good performance comes with iron-clad guarantees. In cryptography, however, a situation where the eavesdropper can easily decode half of all the possible messages is unacceptable —even if the complexity of decoding the rest may be exponential. It is clear that worst-case complexity is an inappropriate criterion in cryptography.

A definition of one-way functions that is closer to what we need in cryptography would replace requirement (iii)—that inverting is worst-case difficult—by a stronger requirement, *that there be no integer k, and no algorithm which, for large enough n, in time $\mathcal{O}(n^k)$ successfully computes $f^{-1}(y)$ for at least $\frac{2^n}{n^k}$ strings y of length n.* That is, there is no polynomial-time algorithm that successfully inverts f on a polynomial fraction of the inputs of length n.

Even this definition is not strong enough, since it assumes that f is to be inverted by a deterministic algorithm. We should allow randomized algorithms, since the eavesdropper may very well use one. We should even allow *non-uniform families of circuits* (recall Section 11.4; in fact, ones with some of the inputs random so that we retain the power of randomization), since in practice an attack on a cryptosystem could focus only on the currently used key size (and could thus invest massive amounts of computation for constructing a circuit that works for the currently used key size).

But even such a "strong" one-way function would not be of immediate cryptographic use. For example, f_{MULT} and f_{EXP} (Examples 12.1 and 12.2) are suspected of being such strong one-way functions, and still they could not be used directly as the basis of a public-key cryptosystem. Fortunately, we had better luck with f_{RSA}. What are the additional features of f_{RSA}, besides properties (i), (ii), and hopefully (iii), that are useful in the cryptographic application?

Recall that $f_{\text{RSA}}(x, e, p, C(p), q, C(q)) = (x^e \bmod pq, e, pq)$ if p and q are primes correctly certified by $C(p)$ and $C(q)$ respectively, and e is prime to $pq - p - q + 1$; if any one of these conditions is violated, then f_{RSA} is essentially undefined (it outputs something useless, like its input). If primes were extremely rare, then it could be difficult to identify input values for which f_{RSA} outputs something meaningful. Obviously, such a one-way function would be useless for cryptography.

Fortunately, primes are relatively abundant. Among all n-bit numbers, about one in $\ln 2 \cdot n$ is a prime (see 11.5.27; and yes, that was a natural logarithm!). If we sample n-bit integers at random, and test them for primality by a randomized algorithm, one expects to hit two primes very soon. Generating *certified* primes is not much harder. Finding a residue e prime to $pq - p - q + 1$ is also easy—there is an abundance of such numbers. We conclude that it is fairly easy to find inputs for which f_{RSA} is "defined."

There is a final important positive property that f_{RSA} has: There is a polynomially-computable function d, with the same inputs as f_{RSA}, that *makes the inversion problem easy.* That is, although there is apparently no fast way to recover $(x, e, p, C(p), q, C(q))$ from $(x^e \bmod pq, pq)$, if we are also given $d(x, e, p, C(p), q, C(q)) = e^{-1} \bmod pq - p - q + 1$ then we can easily invert f_{RSA} —by computing $(x^e)^d \bmod pq$ as in the decoding phase of the RSA cryptosystem. That is, we can easily recover the input $X = (x, e, p, C(p), q, C(q))$ from both $f_{\mathrm{RSA}}(X)$ and $d(X)$, but apparently not from $f_{\mathrm{RSA}}(X)$ alone.

To summarize the additional desirable properties of the RSA function, besides (i), (ii), and (iii) of one-way functions, that we identified in this discussion:

(iv) We can efficiently sample the domain of the one-way function;

(v) There is a polynomially computable function d of the input that trivializes the inversion problem.

Notice that these important properties translate to crucial aspects of the RSA cryptosystem: First, by (iv) Bob can generate random public-secret key pairs relatively fast. More importantly, by (v) *he can decode efficiently*—a one-way function that hides Alice's message from Bob would be remarkable, but quite useless for cryptographic purposes. We call a one-way function that has properties (iv) and (v) a *trapdoor function.* With the necessary reservation about property (iii) of one-way functions, f_{RSA} is a trapdoor function.

Randomized Cryptography

Cryptography can be very a tricky and subtle business. There are legitimate reservations even for public-key cryptosystems based on strong one-way functions as defined in the previous subsection. Even though we require that f can be efficiently inverted on only a negligible fraction of all strings, it could be that these strings include a few crucially important strings —such as "ATTACK AT DAWN," "SELL IBM," and "I LOVE YOU." Also, the deterministic nature of the system enables an eavesdropper to notice repetitions of messages, a potentially valuable piece of information.

In fact, there are two very important messages that are always easy to decode: Suppose that Alice and Bob communicate using the RSA public-key cryptosystem, and very often Alice needs to send to Bob a single confidential bit, $b \in \{0, 1\}$. Should Alice encrypt this bit as an ordinary message, $b^e \bmod pq$? Obviously not. Since $b^e = b$ for $b \in \{0, 1\}$, the encrypted message would be

the same as the original message—that is, not encrypted at all! Single bits are always easy to decode.

There is a simple remedy for this last problem. Alice could generate a random integer $x \leq \frac{pq}{2}$, and then transmit to Bob $y = (2x + b)^e \bmod pq$. Bob receives y, and uses his private key to recover $2x + b$: b is the last bit of the decrypted integer. But is this method secure? Obviously, it can be at most as secure as the RSA public-key cryptosystem, since if the eavesdropper could recover $2x + b$ from y, then he would recover b. Furthermore, it is conceivable that from $(2x + b)^e \bmod pq$ and e one can guess the last bit of $2x + b$, at least with high probability of success, *without guessing all of* $2x + b$. As it happens, it can be proved that *this method of encoding a single bit in the last bit of an integer is exactly as secure as the RSA public-key cryptosystem;* that is, any method that guesses b from $(2x + b)^e \bmod pq$ and e with probability of success significantly greater than $\frac{1}{2}$, can be used to break the RSA public-key cryptosystem (see the references in 12.3.4).

But this important fact opens up a very interesting possibility: After all, any message consists of bits. So, Alice could send any message to Bob by breaking it into individual bits, and transmitting these bits one-by-one, using independently drawn random integers x between 0 and $\lfloor \frac{pq}{2} \rfloor - 1$ each time. Admittedly the resulting *randomized public-key cryptosystem* is much slower than the original RSA, which transmits several hundreds of bits at once. The point is that it is much more secure: Arguably, all problems alluded to in the beginning of this subsection (detecting repetitions, luckily recovering crucial messages, etc.) are not present in the randomized public-key cryptosystem.

12.2 PROTOCOLS

In all chapters of this book before the current one, computation was the activity of a machine that solves a problem. There were many variants, subtleties, and complications, but at least the "sociopolitical situation" in terms of actors, motives, goals, and interests was quite straightforward. There was one "noble knight" (the algorithm) engaged in a valiant and single-minded struggle against an obvious "beast" (the complexity of the problem).

The very nature of the current subject, cryptography, indicates a clear departure from that innocence. By definition, cryptography involves two communicating agents, who may have different and conflicting priorities and interests. Furthermore, and more importantly, they communicate in the presence of eavesdroppers with even murkier motives. In this sense, even the simple situation in Figure 12.1 is something more complex than solving a computational problem, where the only goal is to achieve low complexity. It is a *protocol*, that is, a set of interacting computations, sharing inputs and outputs in arbitrarily complex ways. Furthermore, some of these computations are prescribed to be easy, *and for some it is desired to be hard.*

We shall now examine some more elaborate protocols. Besides involving some of the most clever and cute ideas in the field, some of these protocols will be later shown to coincide with important aspects of complexity.

Signatures

Suppose that Alice wants to send Bob *a signed document* x. But what does this mean? Minimally, a signed message $S_{\text{Alice}}(x)$ is a string that contains the information in the original message x, but is modified in a way that unmistakably identifies the sender.

Public-key cryptosystems provide an elegant solution to the electronic signature problem. Suppose that both Alice and Bob have public and private keys e_{Alice}, d_{Alice}, e_{Bob}, d_{Bob}; as always the public keys are known to the general public, while a private key is known only to its owner. We assume they both use the same encoding and decoding function, E and D. Alice signs x as:

$$S_{\text{Alice}}(x) = (x, D(d_{\text{Alice}}, x))$$

That is, Alice sends x, juxtaposed with a message which is x, *decrypted as if it were an encrypted message received by Alice*. Naturally, if privacy is also desired, the whole signed document can now be encrypted using Bob's public key.

Bob, upon receipt of $S_{\text{Alice}}(x)$, takes the second part $D(d_{\text{Alice}}, x)$ and encodes it, using Alice's public key. We have the following equations:

$$E(d_{\text{Alice}}, D(d_{\text{Alice}}, x)) = D(d_{\text{Alice}}, E(e_{\text{Alice}}, x)) = x.$$

The second equation above is the basic property of any cryptosystem: Decoding inverts encoding. The first equation is the manifestation of a more sophisticated property that certain public-key cryptosystems, including the one based on f_{RSA}, have: *Commutativity*. Commutativity means that, if one encodes a decoded message, one obtains the original message—just like when one decodes an encoded message. The RSA cryptosystem is clearly commutative, since

$$D(d, E(e, x)) = (x^e)^d \bmod pq = (x^d)^e \bmod pq = E(e, D(d, x)).$$

Now Bob, having checked that the second part of the message, when encoded by Alice's public key, is the same as the first, is sure that the message originated with Alice. Because only Alice has the secret key d_{Alice} necessary to produce $D(d_{\text{Alice}}, x)$ from x. And this demonstration can be repeated by Bob, perhaps at some "electronic court of law," to prove that nobody but Alice could have sent the message. Bob can argue plausibly† that nobody can generate $D(d_{\text{Alice}}, x)$ without having Alice's secret key. But in that case Bob could have taken advantage of Alice in much more direct ways than forging her signature...

† And this is the weakest part of this syllogism.

Mental Poker

Suppose that Alice and Bob have agreed upon three n-bit numbers $a < b < c$—the *cards*. They want to randomly choose one card each, so that the following holds: (i) Their cards are different. (ii) All six pairs of distinct cards are equiprobable as outcomes. (iii) Alice's card is known to Alice but not not known to Bob, until Alice decides to announce it, and similarly for Bob. (iv) Since the person with the highest card wins the game, the outcome should be indisputable. An electronic court could, upon inspection of the record of the protocol, be convinced that the parties correctly arrived at the actual outcome.

This improbable feat (whose generalization to fifty-two cards and hands of five is rather obvious) can be achieved by cryptographic techniques. First, the two players agree on *a single large prime number* p, and each has *two secret keys*, an encryption key $e_{\text{Alice}}, e_{\text{Bob}}$ and a decryption key $d_{\text{Alice}}, d_{\text{Bob}}$. We require that $e_{\text{Alice}}d_{\text{Alice}} = e_{\text{Bob}}d_{\text{Bob}} = 1 \bmod p-1$, so that exponentiation by the encryption key modulo p is inversed by exponentiation by the decryption key.

Alice is the dealer. She encrypts the three cards, and sends to Bob the encrypted messages $a^{e_{\text{Alice}}} \bmod p$, $b^{e_{\text{Alice}}} \bmod p$, and $c^{e_{\text{Alice}}} \bmod p$, in some random order. Bob then picks one of the three messages and returns it to Alice, who decodes it and keeps it as her card. In the absence of any information about e_{Alice}, Bob's selection must be random, and thus Alice gets a truly random card suppose it is b.

Bob then encrypts the two remaining cards a and c with his encryption key to obtain $a^{e_{\text{Alice}}e_{\text{Bob}}} \bmod p$ and $c^{e_{\text{Alice}}e_{\text{Bob}}} \bmod p$ and sends a random permutation of the results to Alice. Notice that Alice cannot determine the values of the two cards, as they are encrypted by Bob. Alice now picks one of these cryptic messages, say $a^{e_{\text{Alice}}e_{\text{Bob}}} \bmod p$, *decodes it with her key* d_{Alice}, and sends the result, say $a^{e_{\text{Alice}}e_{\text{Bob}}d_{\text{Alice}}} = a^{e_{\text{Bob}}} \bmod p$ to Bob, as his card. Bob decrypts it using d_{Bob}, and the protocol terminates. Arguably, the four intricate requirements for this protocol have been satisfied.

Interactive Proofs

A nondeterministic algorithm can be thought of as a simple kind of protocol. Suppose that Alice has at her disposal exponential computing powers, but Bob can only afford polynomial computations. Bob and Alice are given a Boolean expression ϕ. Alice has a vivid interest in convincing Bob that ϕ is satisfiable. If ϕ is satisfiable, Alice can succeed: She finds a satisfying truth assignment, using her exponential computing power, and sends it to Bob. Bob, using his meager computational resources, checks that the given truth assignment satisfies his formula, and is convinced. But if ϕ is not satisfiable, no matter how hard Alice tries, she cannot convince Bob of the opposite: Bob will interpret all her arguments as truth assignments, and reject them as non-satisfying. We can say

that this simple protocol decides SAT.

Suppose now that Bob can use randomization, and suppose further that we are willing to tolerate false positives and negatives with an exponentially small error probability. Then Bob, without any help from Alice, can decide all languages in **BPP** (by running the **BPP** algorithm enough times and taking the majoritarian answer).

True, the last one was hardly a protocol, but it does set up the following important question: Suppose that we can use *both* randomization by Bob *and* the exponential powers of Alice. What languages can we accept then?

Definition 12.3: An *interactive proof system* (A, B) is a protocol between Alice and Bob. Alice runs an exponential-time[†] algorithm A, while Bob operates a polynomial-time randomized algorithm B. The input to the protocol is a string x, known to both algorithms. The two exchange a sequence of messages $m_1, m_2, \ldots m_{2|x|^k}$, where Alice sends the odd-numbered ones, and Bob the even ones. All messages are polynomial in length: $|m_i| \leq |x|^k$. Alice goes first, say.

Formally, the messages are defined as follows: $m_1 = A(x)$—that is, the first message is produced by Alice based on input x alone. Subsequently, and for all $i \leq |x|^k$, $m_{2i} = B(x; m_1; \ldots; m_{2i-1}; r_i)$, and $m_{2i-1} = A(x; m_1; \ldots, m_{2i-2})$. Here r_i is the polynomially long random string used by Bob at the ith exchange. *Notice that Alice does not know* r_i (see 12.3.7 for the significance of this). Each even-numbered message is computed by Bob based on the input, r_i, and all previous messages, while each odd-numbered one is computed by Alice based on the input and all previous messages. Finally, in the last message is $m_{2|x|^k} \in$ {"yes", "no"} Bob signals his final approval or disapproval of the input.

We say that (A, B) *decides a language* L if the following is true for each string x: If $x \in L$ then the probability that x is accepted by (A, B) is at least $1 - \frac{1}{2^{|x|}}$; and if $x \notin L$, then the probability that x is accepted by (A', B), *with any exponential algorithm* A' *replacing* A, is at most $\frac{1}{2^{|x}}$. Notice the strong requirement for acceptance: Since Alice is assumed to have an interest in convincing Bob that $x \in L$, if $x \notin L$ Bob should be convinced (by Alice or a malevolent imposter) very infrequently.

Finally, we denote by **IP** the class of all languages decided by an interactive proof system. □

It is clear from the discussion preceding the definition that **IP** contains **NP**, and also **BPP**. **NP** is the subclass of **IP** in which Bob uses no randomization, while **BPP** is the subclass where Bob ignores Alice's replies. But how much larger than these two classes is **IP**? In a later chapter we shall characterize exactly the amazing power of interactive protocols. For the time being, we

[†] It turns out that Alice only needs polynomial space in order to carry out any interaction of this sort, see Problem 12.3.7.

shall see a clever protocol that establishes that a language, not known to be either in **BPP** or in **NP**, is in **IP**.

Example 12.4: GRAPH ISOMORPHISM is an important problem that has resisted all attempts at classification along the lines **P** and **NP**. We are given two graphs $G = (V, E)$ and $G' = (V, E')$ on the same set of nodes, and we ask whether they are *isomorphic*, that is, whether there is a permutation π of V such that $G' = \pi(G)$, where by $\pi(G)$ we denote the graph $(V, \{[\pi(u), \pi(v)] : [u, v] \in E\})$.

GRAPH ISOMORPHISM is obviously in **NP**, but it is not known to be **NP**-complete or in **P** (or even in **coNP**, or **BPP**). Accordingly, its complement GRAPH NONISOMORPHISM ("Given two graphs, are they non-isomorphic?") is not known to be in **NP**, or in **BPP**. However, we shall now show that GRAPH NONISOMORPHISM is in **IP**.

On input $x = (G, G')$, Bob repeats the following for $|x|$ rounds: At the ith round Bob defines a new graph, which is either G or G'; he generates a random bit b_i, and sets $G_i = G$ if $b_i = 1$, else $G_i = G'$. Then Bob generates a random permutation π_i, and sends $m_{2i-1} = (G, \pi_i(G_i))$ to Alice. Alice checks whether the two graphs received are isomorphic. If they are, her answer is $m_{2i} = 1$; if they are not, it is 0.

Finally, after the $|x|$ rounds, Bob accepts if the vectors $(b_1, \ldots, b_{|x|})$ of his random bits and $(m_2, \ldots, m_{2|x|})$ of Alice's replies are identical. This completes the description of the protocol (A, B) for GRAPH NONISOMORPHISM.

Suppose that G and G' are non-isomorphic. Then this protocol will accept the input, because Bob's ith random bit is one if and only if his ith message contains two isomorphic graphs, if and only if Alice's ith reply is one.

Suppose then that G and G' are isomorphic. At the ith round, Alice receives from Bob two graphs G and $\pi_i(G)$. These graphs are isomorphic no matter what b_i was: If it was one, they are isomorphic because they are copies of G. If it was zero, then again $\pi_i(G)$ is just a permuted copy of G', which is isomorphic to G. The point is that *Alice always sees the same picture, G and a random permutation of G.* And from this uninteresting string, Alice has to guess b_i (she must guess all b_i's correctly if she is to cheat Bob). No matter what clever exponential algorithm A' Alice uses, she cannot. She can be lucky a few times, but the probability that she correctly guesses each b_i is at most $\frac{1}{2^{|x|}}$, as required. □

Zero Knowledge

Protocols make extensive use of cryptography and randomization. Signatures and the "mental poker" protocol use cryptography; interactive proofs use randomization. We shall conclude this chapter with a very interesting protocol that uses both.

Suppose that Alice can color the nodes of a large graph $G = (V, E)$ with three colors, such that no two adjacent nodes have the same color. Since 3-COLORING is an **NP**-complete problem, Alice is very proud and excited, and wants to convince Bob that she has a coloring of G. There is nothing difficult here: Since 3-COLORING is in **NP**, she can simply send her 3-coloring to Bob, essentially using the simple interactive protocol that started the previous subsection. But Alice is also worried that, if Bob finds out from her how to color G, he can announce it the same way to *his* friends, without appropriate credit to Alice's ingenuity. What is required here is a *zero-knowledge proof*, that is, an interactive protocol at the end of which Bob is convinced that with very high probability Alice has a legal 3-coloring of G, *but has no clue about the actual 3-coloring.*

Here is a protocol that can achieve this seemingly impossible task. Suppose that Alice's coloring is $\chi : V \mapsto \{00, 11, 01\}$, that is, the three colors are these three strings of length two. The protocol proceeds in rounds. At each round, Alice carries out the following steps: First she generates a random permutation π of the three colors. Then she generates $|V|$ RSA public-private key pairs, (p_i, q_i, d_i, e_i), one for each node $i \in V$. For each node i she computes the probabilistic encoding (y_i, y_i'), according the the jth RSA system, of the color $\pi(\chi(i))$—the color of i, permuted under π. Suppose that $b_i b_i'$ are the two bits of $\pi(\chi(i))$; then $y_i = (2x_i + b_i)^{e_i} \bmod p_i q_i$ and $y_i' = (2x_i' + b_i')^{e_i} \bmod p_i q_i$, where x_i and x_i' are random integers no greater than $\frac{pq}{2}$. All these computations are private to Alice. Alice reveals to Bob the integers $(e_i, p_i q_i, y_i, y_i')$ for each node $i \in V$. That is, the public part of the RSA systems, and the encrypted colors.

It is now Bob's turn to move. Bob picks at random an edge $[i, j] \in E$, and inquires whether its endpoints have a different color, as they should. Alice then reveals to Bob the secret keys d_i and d_j of the endpoints, allowing Bob to compute $b_i = (y_i^{d_i} \bmod p_i q_i) \bmod 2$, and similarly for b_i', b_j, and b_j', and check that, indeed, $b_i b_i' \neq b_j b_j'$. This concludes the description of a round. Alice and Bob repeat this $k|E|$ times, where k is a parameter representing the desired reliability of the protocol.

Obviously, if Alice has a legal coloring of G, all inquiries of Bob will be satisfied. But what if she does not? If she has no legal coloring, then necessarily at each round there is an edge $[i, j] \in E$ such that $\chi(i) = \chi(j)$, and thus $\pi(\chi(i)) = \pi(\chi(j))$. At each round, Bob has a probability of at least $\frac{1}{|E|}$ of discovering that edge. After $k|E|$ rounds, the probability of Bob finding out that Alice has no legal coloring is at least $1 - e^{-k}$.

What is remarkable about this protocol is that *Bob has learned nothing about Alice's coloring of* G in the process. This can be argued along these lines: Suppose that Alice does have a legal 3-coloring, and the protocol is carried out. What does Bob see at each round, after all? Some randomly generated public keys, some probabilistic encryptions of colors. Then he proposes an edge, he sees

two decryption keys, and finally he finds out the two colors $\pi(\chi(u))$ and $\pi(\chi(v))$. But these colors are permuted versions of the original colors of Alice, and so they are nothing else but *a randomly chosen pair of different colors*. In conclusion, *Bob sees nothing that he could not generate* sitting by himself, flipping a fair coin for polynomial time, without Alice and her 3-coloring. We can conclude that *zero knowledge* was exchanged—in fact, a reasonable definition of zero knowledge goes roughly along the above lines, namely that the interactions in the protocol form a random string drawn from a distribution that was available at the beginning of the protocol.

As a final note, it is handy that the zero-knowledge protocol just described works for 3-COLORING, an **NP**-complete problem. Using reductions, it is possible to conclude that *all problems in* **NP** *have zero-knowledge proofs* (see the references in 12.3.6).

12.3 NOTES, REFERENCES, AND PROBLEMS

12.3.1 Public-key cryptography was a bold idea proposed in

- W. Diffie and M. E. Hellman "New directions in cryptography," *IEEE Trans. on Information Theory, 22*, pp. 664–654, 1976,

while the RSA cryptosystem, the most time-resistant implementation of this idea to date, was proposed in

- R. L. Rivest, A. Shamir, and L. Adleman "A method for obtaining digital signatures and public-key cryptosystems," *C.ACM, 21*, pp. 120–126, 1978.

For a recent review of cryptography and its connections with complexity see

- R. L. Rivest "Cryptography," pp. 717–755 in *The Handbook of Theoretical Computer Science, vol. I: Algorithms and Complexity*, edited by J. van Leeuwen, MIT Press, Cambridge, Massachusetts, 1990.

12.3.2 Trapdoor knapsacks. Another clever way of coming up with a plausible one-way function is based on the **NP**-complete problem KNAPSACK (recall Theorem 9.10), from

- R. C. Merkle and M. E. Hellman "Hiding information and signatures in trapdoor knapsacks," *IEEE Trans. on Information Theory, 24*, pp. 525–530, 1978.

Fix n large integers $a_1, \ldots, a_n$, and consider them to be the public key e. Any n-bit vector x can now be interpreted as a subset X of $\{1, \ldots, n\}$. The encrypted message is then $E(e, x) = \sum_{i \in X} a_i$.

Given $K = E(e, x)$, the sum of several integers, anyone who wants to break this cryptosystem has to solve an instance of KNAPSACK. However, Bob can do it easily: He has two secret large numbers N and m, such that $(N, m) = 1$, and the numbers $a'_i = a_i \cdot m \bmod N$ *grow exponentially fast*, that is, $a'_{i+1} > 2a'_i$.

(a) Show that KNAPSACK is easy to solve with such a'_i's.

Therefore, Bob solves, instead of the instance $(a_1, \ldots, a_n, K)$ of KNAPSACK, the easy instance $(a'_1, \ldots, a'_n, K' = K \cdot m \bmod N)$

(b) Show how Bob can easily recover x from the solution of this easy instance.

The problem is, of course, that instances of KNAPSACK such as these, resulting from exponentially growing instances by multiplication by some $m^{-1} \bmod N$, may be easy to break even without knowing m and N. And they are: Several variants of this scheme have been broken, see

- A. Shamir "A polynomial-time algorithm for breaking the basic Merkle-Hellman cryptosystem," *Proc. 23rd IEEE Symp. on the Foundations of Computer Science*, pp. 142–152, 1982, and
- J. C. Lagarias and A. M. Odlyzko "Solving low-density subset sum problems," *Proceedings of the 24th IEEE Symp. on the Foundations of Computer Science*, pp. 1–10, 1983.

The technique used is the basis reduction algorithm we have seen before:

- A. K. Lenstra, H. W. Lenstra, and L. Lovász "Factoring polynomials with rational coefficients," *Math. Ann, 261*, pp. 515 534, 1982.

12.3.3 Unambiguous machines and the class **UP** were introduced in

- L. G. Valiant "Relative complexity of checking and evaluating," in *Inf. Proc. Letters, 5*, pp. 20–23, 1976.

The connection to one-way functions (Theorem 12.1) is from

- J. Grollman and A. L. Selman "Complexity measures for public-key cryptography," *SIAM J. Comp., 17*, pp. 309–335, 1988. Incidentally, this issue of the *SIAM Journal on Computing* was entirely devoted to papers on Cryptography. See also
- E. Allender "The complexity of sparse sets in P," pp. 1–11 in *Structure in Complexity Theory,* edited by A. L. Selman, Lecture Notes in Comp. Sci. Vol. 223, Springer Verlag, Berlin, 1986, and
- J.-Y. Cai, L. Hemachandra "On the power of parity polynomial time," pp. 229–239 in *Proc. 6th Annual Symp. Theor. Aspects of Computing,* Lecture Notes in Computer Science, Volume 349, Springer Verlag, Berlin,, 1989.

for an interesting generalization of **UP** to machines that are guaranteed to have *polynomially few* accepting computations.

12.3.4 Probabilistic encryption was first proposed in

- S. Goldwasser and S. Micali "Probabilistic encryption, and how to play mental poker keeping secret all partial information," *Proc. 14th ACM Symp. on the Theory of Computing*, pp. 365–377, 1982; retitled "Probabilistic encryption," *J.CSS, 28*, pp. 270–299, 1984.

This paper also contains a formalism of "polynomial-time security" for cryptographic systems, and proves that the proposed probabilistic scheme (which is more general than the one we describe at the last subsection of Section 12.1) is indeed secure—assuming a bit-valued version of a trapdoor function, called a *trapdoor predicate*, exists. That guessing the last bit of an encrypted message in the RSA cryptosystem is as hard as getting the whole message (a fact on which our probabilistic scheme was based) was proved in

- W. B. Alexi, B. Chor, O. Gordreich, and C. P. Schnorr "RSA and Rabin functions: Certain parts are as hard as the whole," *SIAM J. Comp., 17*, pp. 194–209, 1988.

12.3.5 Pseudorandom numbers. There is an obvious connection between cryptography and the generation of pseudorandom numbers: The next bit or number in the pseudorandom sequence should not be predictable from current information, exactly as the original message should not be recoverable from the encryption. Exploiting this connection, Manuel Blum and Silvio Micali devised a pseudorandom bit generator that is provably unpredictable *assuming the discrete logarithm problem (Example 12.2) does not have a polynomial-time algorithm:*

- M. Blum and S. Micali "How to generate cryptographically strong sequences of pseudo-random bits," *SIAM J. Comp., 13*, 4, pp. 851–863, 1984.

This construction was shown by Andrew Yao to have important implications for complexity. Such sequences can pass all "polynomial-time statistical tests for randomness," and can therefore be used to run **RP** algorithms: If indeed the discrete logarithm problem is hard, then **RP** can be simulated in far less than exponential (though not quite polynomial) time; see

- A. C. Yao "Theory and application of trapdoor functions," *Proc. 23rd IEEE Symp. on the Foundations of Computer Science*, pp. , 80–91, 1982.

12.3.6 Signatures were a part of the original public-key idea of Diffie and Hellman (see the reference above), while mental poker was proposed by

- A. Shamir, R. L. Rivest, and L. Adleman "Mental poker," pp. 37–43 in *The Mathematical Gardener*, edited by D. Klarner, Wadsworth, Belmont, 1981,

and zero-knowledge proofs by

- S. Goldwasser, S. Micali, and C. Rackoff "The knowledge complexity of interactive proof systems," *Proc. 17th ACM Symp. on the Theory of Computing*, pp. 291–304, 1985; also, *SIAM J. Comp., 18*, pp. 186–208, 1989.

The zero-knowledge protocol for graph coloring given in Section 12.2 is from

- O. Goldreich, S. Micali, and A. Wigderson "Proofs that yield nothing but their validity, and a methodology of cryptographic protocol design," *Proc. 27th IEEE Symp. on the Foundations of Computer Science*, pp. 174–187, 1986.

It is shown in this paper that all problems in **NP** have zero-knowledge proofs, as a consequence (not at all direct) of the fact that the **NP**-complete graph coloring problem has.

12.3.7 The paper by Goldwasser, Micali, and Rackoff cited above also introduced the interactive proof systems and the class **IP** (Section 12.2). At the same conference, Laci Babai had introduced *Arthur-Merlin games*

- L. Babai "Trading group theory for randomness," *Proc. 17th ACM Symp. on the Theory of Computing*, pp. 421–429, 1985.

In Babai's formulation, Arthur plays a weaker Bob *who has to announce to Merlin his random bits*, while Merlin is as all-powerful as Alice. Apparently, this results in a weaker kind of protocol. For example, how would one recognize GRAPH NONISOMORPHISM (recall Section 12.2) under this regime? In the next conference, it was shown that *the powers of the two kinds of protocols coincide:*

- S. Goldwasser and M. Sipser "Private coins vs. public coins in interactive proof systems," *Proc. 18th ACM Symp. on the Theory of Computing*, pp. 59–68, 1986.

This important result is proved by a clever protocol in which public random bits simulate private ones, by analyzing the probability of acceptance of the private-bit protocol.

Problem: Suppose that Alice has *unbounded* computational resources (as opposed to exponential) to run her algorithm A in the definition of **IP**. Or that she only has *polynomial space*. Show that this does not affect the class **IP**.

12.3.8 There is more (or less?) than meets the eye in cryptographic protocols, at least as described in Section 12.2, in their simplest form. Cryptographic protocols are sometimes plagued by even more subtle shortcomings than those we kept discovering in cryptographic systems. For example, signature schemes have been broken

- G. Yuval "How to swindle Rabin," *Cryptologia, 3*, pp. 187–189, 1979,

and the mental poker scheme in Section 12.2 has been shown to leave some information about the cards unhidden

- R. J. Lipton "How to cheat in mental poker," in *Proc. AMS Short Course in Cryptography*, AMS, Providence, 1981.

12.3.9 Problem: Consider the following algorithm for 3-COLORING:

> If G has a clique of size four reply "G is not 3-colorable."
> Otherwise, try all possible 3-colorings of the nodes."

(a) Suppose that all graphs with n nodes are equally probable. Show that the probability that the algorithm executes the second line is 2^{-cn^2} for some $c > 0$.

(b) Conclude that this is a *polynomial average-case algorithm* for the **NP**-complete problem 3-COLORING. For a similar result for 3SAT, see

- E. Koutsoupias and C. H. Papadimitriou "On the greedy heuristic for satisfiability," *Inf. Proc. Letters, 43*, pp. 53–55, 1992.

12.3.10 Average-case complexity. Throughout this book we took a *worst-case approach* to complexity. All our negative results imply that problems are hard *in the worst case*. As we have argued in this chapter, this is not useful evidence of complexity for cryptographic applications. There are many **NP**-complete problems, and natural probabilistic distributions of their instances, for which algorithms exist that solve them in polynomial expected time (see the previous problem).

What kind of complexity-theoretic evidence can identify problems which cannot be solved efficiently on the average (and thus are potentially useful for cryptography)? Leonid Levin proposed a very nice framework for this in

- L. A. Levin "Problems complete in 'average' instance," *Proc. 16th ACM Symposium on the Theory of Computing* p. 465 (yes, one page!), 1984.

Let μ be a *probability distribution over* Σ^*, that is, a function assigning to each string a positive real number, and such that $\sum_{x \in \Sigma^*} \mu(x) = 1$. A *problem* now is not just a language $L \subseteq \Sigma^*$, but a *pair* (L, μ).

We shall only consider distributions that are *polynomially computable*. We say that μ is polynomially computable if its cumulative distribution $M(x) = \sum_{y \leq x} \mu(y)$ can be computed in polynomial time (where the sum is taken over all strings y that are lexicograpgically smaller than x). Notice that the exponential summation must be computed in polynomial time; despite this, most natural distributions (such as random graphs with a fixed edge probability, random strings with all strings of the same length equiprobable, and so on) have this property. Usually, natural distributions of graphs and other kinds of instances only discuss how instances of a particular size

(say, graphs with n nodes) are distributed. Any such distribution can be transformed in the current framework by multiplying the probability of all instances of size n by $\frac{1}{n^2}$, say, and then normalize by $\sum_{i=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}$.

We must first define the class of "satisfactorily solved" problems. We say that problem (L, μ) can be solved in *average polynomial time* if there is a Turing machine M and an integer $k > 0$ such that, if $T_M(x)$ is the number of steps carried out by M on input x, we have

$$\sum_{x \in \Sigma^*} \mu(x) \frac{(T_M(x))^{\frac{1}{k}}}{|x|} < \infty.$$

This peculiar definition is very well motivated: It is both model-independent (recall *left polynomial composition* in Problem 7.4.4), and so it should not be affected if $T_M(x)$ is raised to any constant power (k will be appropriately increased). Also, to be closed under reductions (this is right polynomial composition), it should not be affected if $|x|$ is replaced by a power. And of course, it captures average case complexity under the distribution μ.

A *reduction* from problem (L, μ) to (L', μ') is a reduction R from L to L', with the following additional property: There is an integer $\ell > 0$ such that for all strings x,

$$\mu'(x) \geq \frac{1}{|x|^\ell} \sum_{y \in R^{-1}(x)} \mu(y).$$

That is, we require that the target distribution μ' not be anywhere more than polynomially smaller than the distribution induced by μ and R.

(a) Show that reductions compose.

(b) Show that, if there is a reduction from (L, μ) to (L', μ'), and (L', μ') can be solved in average polynomial time, then (L, μ) can also be solved in average polynomial time.

We say that a problem (L, μ) is *average-case* **NP**-*complete* (Levin's term was "random **NP**-complete") if all problems (L', μ') with $L' \in \mathbf{NP}$ and μ' computable, reduce to it (and of course $L \in \mathbf{NP}$ and μ is computable as well).

Levin's paper contained a completeness result for a "random tiling" problem (see Problem 20.2.10), with a natural distribution. There have been some other completeness results reported, see for example

- Y. Gurevich "The matrix decomposition problem is complete for the average case," *Proc. 31st IEEE Symp. on the Foundations of Computer Science*, pp. 802–811, 1990, and
- R. Venkatesan and S. Rajogopalan, "Average case intractability of matrix and Diophantine problems" *Proc. 24th ACM Symp. on the Theory of Computing*, pp. 632–642, 1992

for some algebraic and number-theoretic complete problems.

CHAPTER

13 APPROXIMABILITY

Although all NP-complete problems share the same worst-case complexity, they have little else in common. When seen from almost any other perspective, they resume their healthy, confusing diversity. Approximability is a case in point.

13.1 APPROXIMATION ALGORITHMS

An **NP**-completeness proof is typically the first act of the analysis of a computational problem by the methods of the theory of algorithms and complexity, not the last. Once **NP**-completeness has been established, we are motivated to explore possibilities that are less ambitious than solving the problem exactly, efficiently, every time. If we are dealing with an optimization problem, we may want to study the behavior of *heuristics*, "quick-and-dirty" algorithms which return feasible solutions that are not necessarily optimal. Such heuristics can be empirically valuable methods for attacking an **NP**-complete optimization problem even when nothing can be proved about their worst-case (or expected) performance. In some fortunate cases, however, the solutions returned by a polynomial-time heuristic are guaranteed to be "not too far from the optimum." We formalize this below:

Definition 13.1: Suppose that A is an optimization problem; this means that for each instance x we have a set of *feasible solutions*, call it $F(x)$, and for each such solution $s \in F(x)$ we have a positive integer cost $c(s)$ (we use the term *cost* and the notation $c(s)$ even in the case of maximization problems). The optimum cost is defined then as $\mathrm{OPT}(x) = \min_{s \in F(x)} c(s)$ (or $\max_{s \in F(x)} c(s)$, if A is a maximization problem). Let M be an algorithm which, given any

instance x, returns a feasible solution $M(x) \in F(x)$. We say that M is an *ϵ-approximation algorithm*, where $\epsilon \geq 0$, if for all x we have

$$\frac{|c(M(x)) - \mathrm{OPT}(x)|}{\max\{\mathrm{OPT}(x), c(M(x))\}} \leq \epsilon.$$

Recall that all costs are assumed to be positive, and thus this ratio is always well-defined. Thus, a heuristic is ϵ-approximate if, intuitively, the "relative error" of the solution found is at most ϵ. We use $\max\{\mathrm{OPT}(x), c(M(x))\}$ in the denominator, instead of the more natural $\mathrm{OPT}(x)$, in order to make the definition symmetric with respect to minimization and maximization problems: This way, for both kinds of problems ϵ takes values between 0 and 1. For maximization problems, an ϵ-approximate algorithm returns solutions that are never smaller than $1 - \epsilon$ times the optimum. For minimization problems, the solution returned is never more than $\frac{1}{1-\epsilon}$ times the optimum. □

For each **NP**-complete optimization problem A we shall be interested in determining the smallest ϵ for which there is a *polynomial-time* ϵ-approximation algorithm for A. Sometimes no such smallest ϵ exists, but there are approximation algorithms that achieve arbitrarily small error ratios (we see an example in the next section).

Definition 13.2: The *approximation threshold* of A is the greatest lower bound of all $\epsilon > 0$ such that there is a polynomial-time ϵ-approximation algorithm for A. □

The approximation threshold of an optimization problem, minimization *or* maximization, can be anywhere between zero (arbitrarily close approximation is possible) and one (essentially no approximation is possible). Of course, for all we know $\mathbf{P} = \mathbf{NP}$, and thus all optimization problems in **NP**, have approximation threshold zero. It turns out that **NP**-complete optimization problems behave in very diverse and intriguing ways with respect to this important parameter because reductions typically fail to preserve the approximation threshold of problems. We turn immediately to some examples.

Node Cover

NODE COVER (Corollary 2 to Theorem 9.4) is an **NP**-complete minimization problem, where we seek the smallest set of nodes $C \subseteq V$ in a graph $G = (V, E)$, such that for each edge in E at least one of its endpoints is in C.

What is a plausible heuristic for obtaining a "good" node cover? Here is a first try: If a node v has high degree, it is obviously useful for covering many edges, and so it is probably a good idea to add it to the cover. This suggests the following "greedy" heuristic:

Start with $C = \emptyset$. While there are still edges left in G, choose the node in G with the largest degree, add it to C, and delete it from G.

As it turns out, this heuristic is *not* an ϵ-approximation algorithm, for *any* $\epsilon < 1$—its error ratio grows as $\log n$ (see Problem 13.4.1), where n is the of the number of nodes of G, and thus no ϵ smaller than 1 is valid.

In order to achieve a decent approximation of NODE COVER, we must employ a technique that appears even less sophisticated than the greedy heuristic:

Start with $C = \emptyset$. While there are still edges left in G, choose any edge $[u, v]$, add both u and v to C, and delete them from G.

Suppose that this heuristic ends up with a node cover C. How far off the optimum can C be? Notice that C contains $\frac{1}{2}|C|$ edges of G, no two of which share a node (a *matching*). Any node cover, *including the optimum one*, must contain at least one node from each of these edges (otherwise, an edge would not be covered). It follows that $\mathrm{OPT}(G) \geq \frac{1}{2}|C|$, and thus $\frac{|C| - \mathrm{OPT}(G)}{|C|} \leq \frac{1}{2}$. We have shown the following:

Theorem 13.1: The approximation threshold of NODE COVER is at most $\frac{1}{2}$. □

Surprisingly, this simple algorithm is the best approximation algorithm known for NODE COVER.

Maximum Satisfiability

In MAXSAT we are given a set of clauses, and we seek the truth assignment that satisfies the most. The problem is **NP**-complete even if the clauses have at most two literals (recall Theorem 9.2).

Our approximation algorithm for MAXSAT is best described in terms of a more general problem called k-MAXGSAT (for maximum *generalized* satisfiability). In this problem we are given a set of Boolean expressions $\Phi = \{\phi_1, \ldots, \phi_m\}$ in n variables, where each expression is not necessarily a disjunction of literals as in MAXSAT, but is a general Boolean expression involving at most k of the n Boolean variables, where $k > 0$ is a fixed constant (we can in fact assume for simplicity that each expression involves *exactly* k variables, some of which may not be explicitly mentioned in it). We are seeking the truth assignment that satisfies the most expressions.

Although our approximation algorithm for this problem will be perfectly deterministic, it is best motivated by a probabilistic consideration. Suppose that we pick one of the 2^n truth assignments at random. How many expressions in Φ should we expect to satisfy? The answer is easy to calculate. Each expression $\phi_i \in \Phi$ involves k Boolean variables. Out of the 2^k truth assignments, we can easily calculate the number t_i of truth assignments that satisfy ϕ_i. Thus, a random truth assignment will satisfy ϕ_i with probability $p(\phi_i) = \frac{t_i}{2^k}$. The expected number of satisfied expressions is simply the sum of these probabilities: $p(\Phi) = \sum_{i=1}^{m} p(\phi_i)$.

Suppose that we set $x_1 =$ **true** in all expressions of Φ; a set of expressions

$\Phi[x_1 = \textbf{true}]$ involving the variables $x_2, \ldots, x_n$ results, and we can again calculate $p(\Phi[x_i = \textbf{true}])$. Similarly for $p(\Phi[x_1 = \textbf{false}])$. Now it is very easy to see that

$$p(\Phi) = \frac{1}{2}(p(\Phi[x_1 = \textbf{true}]) + p(\Phi[x_1 = \textbf{false}])).$$

This equation means that, if we modify Φ by setting x_1 equal to the truth value t that yields the largest $p(\Phi[x_1 = t])$, *we end up with an expression set with expectation at least as large as the original.*

We can continue like this, always assigning to the next variable the value that maximizes the expectation of the resulting expression set. In the end, all variables have been given values, and all expressions are either **true** (have been satisfied) or **false** (have been falsified). However, since our expectation never decreased in the process, we know that at least $p(\Phi)$ expressions have been satisfied.

Thus, our algorithm satisfies at least $p(\Phi)$ expressions. Since the optimum cannot be more than the total number of expressions in Φ that are *individually satisfiable* (that is, those for which $p(\phi_i) > 0$), the ratio is at least equal to the smallest positive $p(\phi_i)$ -recall that $p(\Phi)$ is the sum of all these positive $p(\phi_i)$'s. We conclude that the above heuristic is a polynomial-time ϵ-approximation algorithm for k-MAXGSAT, *where ϵ is one minus the smallest probability of satisfaction of any satisfiable formula in Φ.* For any satisfiable expression ϕ_i involving k Boolean variables, this probability is at least 2^{-k} (since at least one of the 2^k possible truth assignments on the k variables must satisfy the expression) and thus this algorithm is ϵ-approximate with $\epsilon = 1 - 2^{-k}$.

Now if the ϕ_i's are clauses (this brings us back to MAXSAT), then the situation is far better: The probability of satisfaction is at least $\frac{1}{2}$, and $\epsilon = \frac{1}{2}$. If we restrict the clauses to have *at least* k distinct literals (notice the reversal in the usual restriction), then the probability that a random truth assignment satisfies a clause is obviously $1-2^{-k}$ (all truth assignments are satisfying, except for the one that makes all literals **false**), and the approximation ratio becomes $\epsilon = 2^{-k}$.

We can summarize our discussion of maximum satisfiability problems thus:

Theorem 13.2: The approximation threshold of k-MAXGSAT is at most $1-2^{-k}$. The approximation threshold of MAXSAT (the special case of MAXGSAT where all expressions are clauses) is at most $\frac{1}{2}$; and when each clause has at least k distinct literals, the approximation threshold of the resulting problem is at most 2^{-k}. □

These are the best polynomial-time approximation algorithms known for k-MAXGSAT and MAXSAT with at least k literals per clause; the best upper bound known for the approximation threshold of general MAXSAT is $\frac{1}{4}$.

Maximum Cut

In MAX-CUT we want to partition the nodes of $G = (V, E)$ into two sets S and $V - S$ such that there are as many edges as possible between S and $V - S$; MAX-CUT is **NP**-complete (Theorem 9.5).

An interesting approximation algorithm for MAX-CUT is based on the idea of *local improvement* (recall Example 10.6). We start from any partition of the nodes of $G = (V, E)$ (even $S = \emptyset$), and repeat the following step: If the cut can be made larger (more edges would be in it) by adding a single node to S, or by deleting a single node from S, then we do so. If no improvement is possible, we stop and return the cut thus obtained.

One can develop such local improvement algorithms for just about any optimization problem. Sometimes such heuristics are extremely useful, but usually very little can be proved about their performance —both the time required (recall Example 10.6) and the ratio to the optimum. Fortunately, the present case is an exception. First notice that, since the maximum cut can have at most $|E|$ edges, and each local improvement adds at least one edge to the cut, the algorithm must end after at most $|E|$ improvements. (In general, any local improvement algorithm in an optimization problem with polynomially bounded costs will be polynomial.) Furthermore, *we claim that the cut resulting from this algorithm has at least half as many edges as the optimum*, and thus this simple local improvement heuristic is a polynomial-time $\frac{1}{2}$-approximation algorithm for MAX-CUT.

In proof, consider a decomposition of V into four disjoint subsets $V = V_1 \cup V_2 \cup V_3 \cup V_4$, such that the partition obtained by our heuristic is $(V_1 \cup V_2, V_3 \cup V_4)$, whereas the optimum partition is $(V_1 \cup V_3, V_2 \cup V_4)$. Let e_{ij}, with $1 \leq i \leq j \leq 4$ be the number of edges between node sets V_i and V_j (see Figure 13.1). All we know about our partition is that it cannot be improved by migrating any node in the other set. Thus, for each node in V_1, its edges to V_1 and V_2 are outnumbered by those to V_3 and V_4. Considering now all nodes in V_1 together we obtain $2e_{11} + e_{12} \leq e_{13} + e_{14}$, from which we conclude $e_{12} \leq e_{13} + e_{14}$ Similarly we can get the following inequalities, by considering the other three sets of nodes:

$$e_{12} \leq e_{23} + e_{24}$$
$$e_{34} \leq e_{23} + e_{13}$$
$$e_{34} \leq e_{14} + e_{24}.$$

Adding all these inequalities, dividing both sides by two, and adding the inequality $e_{14} + e_{23} \leq e_{14} + e_{23} + e_{13} + e_{24}$ we obtain

$$e_{12} + e_{34} + e_{14} + e_{23} \leq 2 \cdot (e_{13} + e_{14} + e_{23} + e_{24}),$$

which is the same as saying that our solution is at least half the optimum. We have shown:

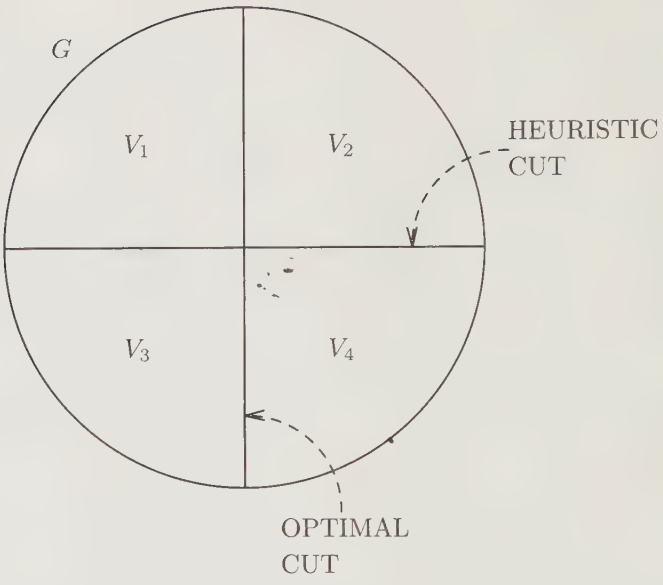


Figure 13-1. The argument for MAX-CUT.

Theorem 13.3: The approximation threshold of MAX-CUT is at most $\frac{1}{2}$. □

The Traveling Salesman Problem

In all three cases of NODE COVER, MAXSAT, and MAX-CUT that we have seen so far, we have exhibited algorithms that guarantee some approximation threshold strictly less than one. For the TSP the situation is very bleak in comparison: If there is a polynomial-time ϵ-approximation algorithm for the TSP for any $\epsilon < 1$, then it follows that **P** = **NP** and approximation algorithms are pointless...

Theorem 13.4: Unless **P** = **NP**, the approximation threshold of the TSP is one.

Proof: Suppose that there is a polynomial-time ϵ-approximation algorithm for the TSP for some $\epsilon < 1$. Using this algorithm, we shall derive a polynomial-time algorithm for the **NP**-complete problem HAMILTON CYCLE (recall the Theorem 9.7 and Problem 9.5.15). Notice that this would conclude the proof.

Given any graph $G = (V, E)$, our algorithm for HAMILTON CYCLE constructs an instance of the TSP with $|V|$ cities. The distance between city i and j is one if there is an edge between nodes i and j in G, and it is $\frac{|V|}{1-\epsilon}$ if there is no $[i, j]$ edge in E. Having constructed this instance of the TSP, we next apply our hypothetical polynomial-time ϵ-approximation algorithm to it. There are two cases: If the algorithm returns a tour of total cost $|V|$ —that is, with only

unit-length edges then we know that G has a Hamilton cycle. If on the other hand the algorithm returns a tour with at least one edge of length $\frac{|V|}{1-\epsilon}$, then the total length of this tour is strictly greater than $\frac{|V|}{1-\epsilon}$. Since we have assumed that our algorithm is ϵ-approximate, that is, the optimum is never less than $1-\epsilon$ times the solution returned, we must conclude that the optimum tour has cost greater than $|V|$, and thus G has no Hamilton cycle. Thus, we can decide whether a graph has a Hamilton cycle simply by creating the instance of the TSP as described, and running the hypothetical ϵ-approximate algorithm on it. □

Notice the specialized kind of reduction employed in this proof of impossibility: In the constructed instance there is a large "gap" between the optimum cost when the original instance is a "yes" instance of the Hamilton cycle problem, and the optimum cost when the original instance is a "no" instance. It is then shown that an approximation algorithm could detect such a gap.

As usual, a negative result for a problem may not hold for its special cases. Let us consider the special case of the TSP in which all distances are either 1 or 2 (this is the special case that we proved **NP**-complete in Corollary to Theorem 9.7). It is amusing to notice that in this case, *any* algorithm is $\frac{1}{2}$-approximate because all tours have length at most twice the optimum! But we can do much better: There is a polynomial-time $\frac{1}{7}$-approximation algorithm for this problem (see the references in 13.4.8). Even in the more general case in which the distances are not quite all ones and twos, but they *satisfy the triangle inequality* $d_{ij} + d_{jk} \leq d_{ik}$, there is a very simple and clever polynomial-time $\frac{1}{3}$-approximation algorithm (see the references in 13.4.8). In both cases, we know of no better approximation algorithms.

Knapsack

We have seen several optimization problems (MAXSAT, NODE COVER, MAX-CUT, TSP with distances 1 and 2) for which ϵ-approximation exists for some ϵ (and it is open whether smaller ϵ's are achievable), and the general TSP for which no ϵ-approximation is possible unless $\mathbf{P} = \mathbf{NP}$. KNAPSACK is an optimization problem for which approximability has no limits:

Theorem 13.5: The approximation threshold of KNAPSACK is 0. That is, for any $\epsilon > 0$ there is a polynomial ϵ-approximation algorithm for KNAPSACK.

Proof: Suppose that we are given an instance x of KNAPSACK. That is, we have n weights $w_i, i = 1, \ldots, n$, a weight limit W, and n values $v_i, i = 1, \ldots, n$. We must find a subset $S \subseteq \{1, 2, \ldots, n\}$ such that $\sum_{i \in S} w_i \leq W$ and $\sum_{i \in S} v_i$ is the largest possible.

We have seen in Section 9.4 that there is a *pseudopolynomial* algorithm for KNAPSACK, one that works in time proportional to *the weights* in the instance. We develop now a *dual* approach, one that works with the values instead of the

weights. Let $V = \max\{v_1, \ldots, v_n\}$ be the maximum value, and let us define for each $i = 0, 1, \ldots, n$ and $0 \leq v \leq nV$ the quantity $W(i, v)$ to be *the minimum weight attainable by selecting some among the i first items, so that their value is exactly v.* We start with $W(0, v) = \infty$ for all i and v, and then

$$W(i+1, v) = \min\{W(i, v), W(i, v - v_{i+1}) + w_{i+1}\}.$$

In the end, we pick the largest v such that $W(n, v) \leq W$. Obviously, this algorithm solves KNAPSACK in time $\mathcal{O}(n^2V)$.

But of course the values may be huge integers, and this algorithm is a pseudopolynomial, not a polynomial one. However, now that we are only interested in *approximating* the optimum value, a maneuver suggests itself: We may want to disregard the last few bits of the values, thus *trading off accuracy for speed.* Given the instance $x = (w_1, \ldots, w_n, W, v_1, \ldots, v_n)$, we can define the *approximate* instance $x' = (w_1, \ldots, w_n, W, v'_1, \ldots, v'_n)$, where the new values are $v'_i = 2^b \lfloor \frac{v_i}{2^b} \rfloor$, the old values with their b least significant bits replaced by zeros; b is an important parameter to be fixed in good time.

If we solve the approximate instance x' instead of x, the time required will be only $\mathcal{O}(\frac{n^2V}{2^b})$, because we can ignore the trailing zeros in the v'_i's. The solution S' obtained will in general be different from the optimum solution S of x, but the following sequence of inequalities shows that their values cannot be very far:

$$\sum_{i \in S} v_i \geq \sum_{i \in S'} v_i \geq \sum_{i \in S'} v'_i \geq \sum_{i \in S} v'_i \geq \sum_{i \in S} (v_i - 2^b) \geq \sum_{i \in S} v_i - n2^b$$

The first inequality holds because S is optimum in x, the second because $v'_i \leq v_i$, the next because S' is optimum in x', the next because $v'_i \geq v_i - 2^b$, and the last because $|S| \leq n$. Comparing the second and last expression, we conclude that the value of the solution returned by our approximation algorithm is at most $n2^b$ below the optimum. Since V is a lower bound on the value of the optimum solution (assume with no loss of generality that that $w_i \leq W$ for all i), the relative deviation from the optimum is at most $\epsilon = \frac{n2^b}{V}$.

We are at a remarkably favorable position: Given *any* user-supplied $\epsilon > 0$, we can truncate the last $b = \lceil \log \frac{\epsilon V}{n} \rceil$ bits of the values, and arrive at an ϵ-approximation algorithm with running time $\mathcal{O}(\frac{n^2V}{2^b}) = \mathcal{O}(\frac{n^3}{\epsilon})$ — a polynomial! We conclude that there is a polynomial-time ϵ-approximation algorithm for any $\epsilon > 0$. Consequently, the greatest lower bound of all achievable ratios is zero. □

Any problem with zero approximation threshold, such as KNAPSACK, has a sequence of algorithms whose error ratios have limit 0. In the case of KNAPSACK the sequence is especially well-behaved, in that the algorithms in the sequence can be seen as the same algorithm supplied with different ϵ's.

Definition 13.3: A *polynomial-time approximation scheme* for an optimization problem A is an algorithm which, for each $\epsilon > 0$ and instance x of A, returns a solution with a relative error of at most ϵ, in time which is bounded by a polynomial (depending on ϵ) of $|x|$. In the case of KNAPSACK, where the polynomial depends *polynomially* on $\frac{1}{\epsilon}$ as well—recall the $\mathcal{O}(\frac{n^3}{\epsilon})$ bound the approximation scheme is called *fully polynomial.* □

Not all polynomial-time approximation schemes need be fully polynomial. For example, no strongly **NP**-complete optimization problem can have a fully polynomial-time approximation scheme, unless **P** = **NP** (see Problem 13.4.2). For example, there is a polynomial-time approximation scheme for BIN PACKING (which is strongly **NP**-complete, recall Theorem 9.11) whose time bound depends exponentially on $\frac{1}{\epsilon}$ (see Problem 13.4.6). For another example of such a scheme, see the next subsection.

Maximum Independent Set

We have seen optimization problems all over the approximability spectrum. Some problems such as the TSP have approximation threshold one (unless **P** = **NP**); others like KNAPSACK have approximation threshold zero; and still others (NODE COVER, MAXSAT, etc.) seem to be in between, with an approximation threshold which is known to be strictly less than one, but not known to be zero. We shall next prove that the INDEPENDENT SET problem belongs in one of the two extreme classes: *Its approximation threshold is either zero or one* (later in this chapter we shall see which of the two extremes is the true answer).

This result is shown by a *product construction.* Let $G = (V, E)$ be a graph. The *square* of G, G^2, is a graph with vertices $V \times V$, and the edges $\{[(u, u'), (v, v')] :$ either $u = v$ and $[u', v'] \in E$, or $[u, v] \in E\}$. See Figure 13.2 for an example. The crucial property of G^2 is this:

Lemma 13.1: G has an independent set of size k if and only if G^2 has an independent set of size k^2.

Proof: If G has an independent set $I \subseteq V$ with $|I| = k$, then the following is an independent set of G^2 of size k^2: $\{(u, v) : u, v \in I\}$. Conversely, if I^2 is an independent set of G^2 with k^2 vertices, then both $\{u : (u, v) \in I^2$ for some $v \in V\}$ or $\{v : (u, v) \in I^2$ for some $u \in V\}$ are independent sets of G, and one of them must have at least k vertices. □

From this we can show:

Theorem 13.6: If there is an ϵ_0-approximation algorithm for INDEPENDENT SET for *any* $\epsilon_0 < 1$, then there is a polynomial-time approximation scheme for INDEPENDENT SET.

Proof: Suppose that a $\mathcal{O}(n^k)$ time-bounded ϵ_0-approximation algorithm exists, and we are given a graph G. If we apply this algorithm to G^2, we obtain in time

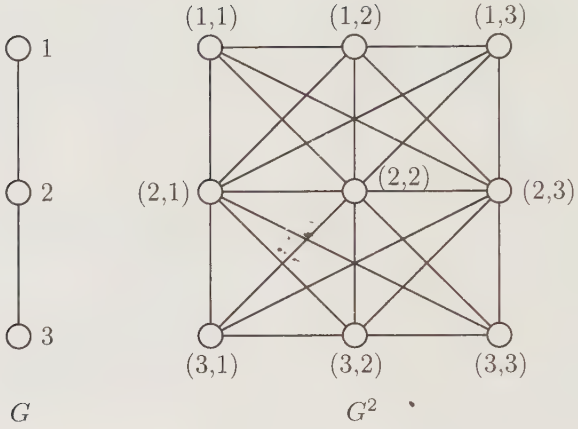


Figure 13-2. The product construction.

$\mathcal{O}(n^{2k})$ an independent set of size at least $(1-\epsilon_0)\cdot k^2$, where k is the maximum independent set of G (and thus k^2 is the maximum independent set of G^2). From this, by the construction in the Lemma, we can get an independent set of G equal to at least the square root of $(1-\epsilon_0)\cdot k^2$, or $\sqrt{1-\epsilon_0}\cdot k$. That is, if we have an ϵ_0-approximation algorithm for INDEPENDENT SET, then we have an ϵ_1-approximation algorithm, where $\epsilon_1 = 1-\sqrt{1-\epsilon_0}$.

Thus, if we have an ϵ_0-approximation algorithm, then the product construction yields an ϵ_1-approximation algorithm. If we apply the product construction twice (that is, apply our approximation algorithm to $(G^2)^2$) then we have an ϵ_2-approximation algorithm, where $1-\epsilon_2 = \sqrt[4]{1-\epsilon_0}$. And so on.

For any given $\epsilon > 0$, however small, we can repeat the product construction $\ell = \lceil \log \frac{\log(1-\epsilon_0)}{\log(1-\epsilon)} \rceil$ times. We obtain an algorithm with time bound $\mathcal{O}(n^{2^\ell k}) = \mathcal{O}(n^{k\frac{\log(1-\epsilon_0)}{\log(1-\epsilon)}})$, and approximation ratio at most ϵ, the desired polynomial-time approximation scheme. (Notice that this time bound is *not* a polynomial in n and $\frac{1}{\epsilon}$, and hence this (hypothetical) polynomial-time approximation scheme would not be fully polynomial.) □

It is instructive at this point to contrast the approximability of the INDEPENDENT SET and NODE COVER problems, two problems that are reducible to each other in a remarkably trivial way (recall Corollary 1 to Theorem 9.3).

Another interesting post scriptum to the approximability of INDEPENDENT SET: If we restrict our graphs so that no node has degree greater than four, then the resulting special case, which we may call k-DEGREE INDEPENDENT SET, remains **NP**-complete (Corollary 1 to Theorem 9.4). However, a simple approximation algorithm is now possible.

We start with $I = \emptyset$. While there are nodes left in G we repeatedly delete from G any node v and all of its adjacent nodes, adding v to I. Obviously, the resulting I will be an independent set of G. Since each stage of the algorithm adds another node to I and deletes at most $k+1$ nodes from G (the node added and its neighbors, at most k of them), the resulting independent set has at least $\frac{|V|}{k+1}$ nodes, which is at least $\frac{1}{k+1}$ times the true maximum independent set. We have shown:

Theorem 13.7: The approximation threshold of the k-DEGREE INDEPENDENT SET problem is at most $\frac{k}{k+1}$. □

Once again, no better polynomial-time approximation algorithm for this problem is known.

13.2 APPROXIMATION AND COMPLEXITY

A polynomial-time approximation scheme for an optimization problem is rightfully considered the next best thing to a polynomial-time *exact* algorithm for the problem. For **NP**-complete optimization problems an important question is whether such a scheme exists. Since individual questions of this sort are so hard to answer, in this section we do something that parallels the development of **NP**-completeness: We lump many such problems together by reductions so that they are all complete for a natural and meaningful complexity class.

L-Reductions

We have seen in several occasions that ordinary reductions are grossly inadequate for studying approximability. We next introduce a careful kind of reduction that preserves approximability.

Definition 13.4: Optimization problems are certainly function problems (since the optimum solution, and not just a "yes" or "no" answer, is sought). Recall from Definition 10.1 that a reduction between function problems A and B is a pair of functions R and S, where R is computable in logarithmic space and S in polynomial time, such that if x is an instance of A then $R(x)$ is an instance of B; and furthermore, if y is an answer of $R(x)$ then $S(y)$ is an answer of x.

Suppose that A and B are optimization problems (maximization *or* minimization). An *L-reduction* from A to B is a pair of functions R and S, both computable in logarithmic space, with the following two additional properties: First, if x is an instance of A with optimum cost $\mathrm{OPT}(x)$, then $R(x)$ is an instance of B with optimum cost that satisfies

$$\mathrm{OPT}(R(x)) \leq \alpha \cdot \mathrm{OPT}(x),$$

where α is a positive constant. Second, if s is any feasible solution of $R(x)$,

then $S(s)$ is a feasible solution of x such that

$$|\mathrm{OPT}(x) - c(S(s))| \leq \beta \cdot |\mathrm{OPT}(R(x)) - c(s)|,$$

where β is another positive constant particular to the reduction (and we use c to denote the cost in both instances). That is, S is guaranteed to return a feasible solution of x which is not much more suboptimal than the given solution of $R(x)$. Notice that, by the second property, an L-reduction is a true reduction: If s is the optimum solution of $R(x)$, then indeed $S(s)$ must the optimum solution of x. □

Example 13.1: The trivial reduction from INDEPENDENT SET to NODE COVER (R is the identity function, returning the same graph G, while S replaces C with $V - C$) is not an L-reduction. Its flaw is that the optimum node cover may be arbitrarily larger than the optimum independent set (consider the case in which G is a large clique), violating the first condition.

However, if we restrict our graphs to have maximum degree k, the problems go away, and (R, S) is indeed an L-reduction from k-DEGREE INDEPENDENT SET to k-DEGREE NODE COVER. In proof, if the maximum degree is k then the maximum independent set is at least $\frac{|V|}{k+1}$, while the minimum node cover has at most $|V|$ nodes, and so the constant $\alpha = k + 1$ satisfies the first condition. And the difference between any cover C and the optimum is the same as the difference between $V - C$ and the maximum independent set, and hence we can take $\beta = 1$ in the second condition.

Similarly, it is easy to see that (R, S) is also an L-reduction in the opposite direction, with the same α and β. □

L-reductions have the important composition property of ordinary reductions (Proposition 8.2):

Proposition 13.1: If (R, S) is an L-reduction from problem A to problem B, and (R', S') is an L-reduction from B to C, then their composition $(R \cdot R', S' \cdot S)$ is an L-reduction from A to C.

Proof: It follows from Proposition 8.2 that $R \cdot R'$ and $S' \cdot S$ are computable in logarithmic space. Also, if x is an instance of A, we have $\mathrm{OPT}(x) \leq \alpha \mathrm{OPT}(R(x))$ and $\mathrm{OPT}(R(x)) \leq \alpha \mathrm{OPT}(R'(R(x)))$, where α and α' are the corresponding constants, and therefore $\mathrm{OPT}(x) \leq \alpha \cdot \alpha' \mathrm{OPT}(R'(R(x)))$ and $R \cdot R'$ satisfies the first condition with the constant $\alpha \cdot \alpha'$. Similarly, it is easy to check that $S' \cdot S$ satisfies the second condition with the constant $\beta \cdot \beta'$. □

The key property of L-reductions is that they preserve approximability:

Proposition 13.2: If there is an L-reduction (R, S) from A to B with constants α and β, and there is a polynomial-time ϵ-approximation algorithm for B, then there is a polynomial-time $\frac{\alpha\beta\epsilon}{1-\epsilon}$-approximation algorithm for A.

Proof: The algorithm is this: Given an instance x of A, we construct the instance $R(x)$ of B, and then apply to it the assumed ϵ-approximation algorithm for B to obtain solution s. Finally, we compute the solution $S(s)$ of A. We claim that this is an $\frac{\alpha\beta\epsilon}{1-\epsilon}$-approximation algorithm for A.

In proof, consider the ratio $\frac{|\mathrm{OPT}(x)-c(S(s))|}{\max\{\mathrm{OPT}(x),c(S(s))\}}$. By the second property of L-reductions, the numerator is at most $\beta|\mathrm{OPT}(R(x)) - c(s)|$. By the first property of L-reductions, the denominator is at least $\frac{\mathrm{OPT}(R(x))}{\alpha}$, which is at least $\frac{\max\{\mathrm{OPT}(R(x)),c(s)\}}{\alpha}(1-\epsilon)$. Dividing the two inequalities we conclude that $\frac{|\mathrm{OPT}(x)-x(S(s))|}{\max\{\mathrm{OPT}(x),c(S(s))\}} \leq \frac{\alpha\beta}{1-\epsilon}\frac{|\mathrm{OPT}(R(x))-c(s)|}{\max\{\mathrm{OPT}(R(x)),c(s)\}}$. The latter quantity is at most $\frac{\alpha\beta\epsilon}{1-\epsilon}$ by our hypothesis about the ϵ-approximation algorithm for B. □

The important property of the expression $\frac{\alpha\beta\epsilon}{1-\epsilon}$ in the statement of Proposition 13.2 is this: *If ϵ tends to zero from positive values, then so does the expression.* This implies the following:

Corollary: If there is an L-reduction from A to B and there is a polynomial-time approximation scheme for B, then there is a polynomial-time approximation scheme for A. □

The Class MAXSNP

Our development of a complexity theory of approximability parallels the reasoning that led us to the **P** $\neq$ **NP** conjecture. In both cases, for several natural problems we asked an important and difficult-to-answer question (then whether they have a polynomial-time algorithm, now whether they have a polynomial-time approximation scheme). We next defined a concept of reduction that preserves the property in question. To proceed we need the analog of **NP**, a broad class of problems, that contains many important complete ones. We define it next.

Definition 13.5: Our motivation for the next definition comes from Fagin's Theorem, stating that all graph-theoretic properties in **NP** can be expressed in existential second-order logic (Theorem 8.3). There is an interesting fragment of **NP**, called *strict* **NP** or **SNP**, which consists of all properties expressible as

$$\exists S \forall x_1 \forall x_2 \ldots \forall x_k \phi(S, G, x_1, \ldots, x_k),$$

where ϕ is a quantifier-free First-Order expression involving the variables x_i and the structures G (the input) and S. **NP** is more general than **SNP**, in that it allows arbitrary first-order quantifiers, not just universal ones.

Naturally, **SNP** contains decision problems, and we are presently interested in defining a class of *optimization* problems. There is a simple and interesting way to obtain a broad class of optimization problems by modifying the **SNP** expressions. Consider any such expression $\exists S \forall x_1 \forall x_2 \ldots \forall x_k \phi$. It asks for a

relation S such that all possible k-tuples of nodes $(x_1, \ldots, x_k)$ satisfy ϕ. Suppose that we compromise a little. Instead of requiring that ϕ hold *for all* k-tuples, we seek the relation S such that ϕ holds for *as many* k-*tuples* $(x_1, \ldots, x_k)$ *as possible*. We thus arrive at an optimization problem. We generalize the situation slightly, so that the input structure G is not necessarily a single binary relation, but a collection $G_1, \ldots, G_m$ of relations of arbitrary arity.

Define now $\mathbf{MAXSNP}_0$ (not yet our final goal) to be the following class of optimization problems: Problem A in this class is defined in terms of the expression

$$\max_S |\{(x_1, \ldots, x_k) \in V^k : \phi(G_1, \ldots, G_m, S, x_1, \ldots, x_k)\}|$$

(compare with $\exists S \forall x_1 \forall x_2 \ldots \forall x_k \phi$). The input to a problem A is a set of relations $G_1, \ldots, G_m$ over a finite universe V. We are seeking a relation $S \subseteq V^r$ such that the number of k-tuples $(x_1, \ldots, x_k)$ for which ϕ holds is as large as possible. Notice how the expression that defines A is derived from the original $\exists S \forall x_1 \forall x_2 \ldots \forall x_k \phi$. The existential quantifier now seeks the maximizing S, while the sequence of k universal quantifiers has become a counter of k-tuples.

Finally, define **MAXSNP** to be the class of all optimization problems that are L-reducible to a problem in $\mathbf{MAXSNP}_0$. □

Example 13.2: The problem MAX-CUT is in $\mathbf{MAXSNP}_0$, and therefore in **MAXSNP**. In proof, MAX-CUT can be written as follows:

$$\max_{S \subseteq V} |\{(x, y) : ((G(x, y) \vee G(y, x)) \wedge S(x) \wedge \neg S(y))\}|.$$

Here we represent a graph as a directed graph, assigning to each undirected edge an arbitrary direction. The problem as stated asks for the subset S of nodes that maximizes the number of edges that enter S or leave S; that is, the maximum cut in the underlying undirected graph.

MAX2SAT (maximizing the number of satisfied clauses, where each clause has two literals) is also in **MAXSNP**. Here we have three input relations, G_0, G_1, and G_2. Intuitively, G_i contains all clauses with i negative literals; that is, $G_0(x, y)$ if and only if $(x \vee y)$ is a clause of the represented expression; $G_1(x, y)$ if and only if $(\neg x \vee y)$ is a clause; and $G_2(x, y)$ if and only if $(\neg x \vee \neg y)$ is a clause. With these somewhat complicated input conventions we can write MAX2SAT as $\max_{S \subseteq V} |\{(x, y) : \phi(G_0, G_1, G_2, S, x, y)\}|$, where ϕ is the following expression:

$$(G_0(x, y) \wedge (S(x) \vee S(y)) \vee (G_1(x, y) \wedge (\neg S(x) \vee S(y)) \vee (G_2(x, y) \wedge (\neg S(x) \vee \neg S(y)),$$

where S now stands for the set of **true** variables. It is not hard to see that the problem, as stated, indeed asks for the truth assignment that maximizes the

total number of satisfied clauses. A similar construction, with four relations, establishes that MAX3SAT is in **MAXSNP**.

For k-DEGREE INDEPENDENT SET, our input is a non-standard representation of a graph $G = (V, E)$ with maximum degree k in terms of a $(k+1)$-ary relation H. H contains the $|V|$ $(k+1)$-tuples $(x, y_1, \ldots, y_k)$ such that the y_i's are the neighbors of node x (with repetitions when x has fewer than k neighbors). The k-DEGREE INDEPENDENT SET problem can be written thus:

$$\max_{S \subseteq V} |\{(x, y_1, \ldots, y_k) : [(x, y_1, \ldots, y_k) \in H] \wedge [x \in S] \wedge [y_1 \notin S] \wedge \ldots \wedge [y_k \notin S]\}|.$$

S is the independent set.

Finally, k-DEGREE NODE COVER is in **MAXSNP** because it L-reduces to k-DEGREE INDEPENDENT SET (recall Example 13.1). Notice that, since $\mathbf{MAXSNP}_0$ contains by definition only maximization problems, an L-reduction to another problem in **MAXSNP** is the only way for a minimization problem, such as k-DEGREE NODE COVER, to be in **MAXSNP**. □

All four maximization problems in **MAXSNP** that we saw in Example 13.2 were shown in the previous section to have a polynomial-time ϵ-approximation algorithm, for some $\epsilon < 1$. This is no coincidence:

Theorem 13.8: Let A be a problem in $\mathbf{MAXSNP}_0$. Suppose that A is of the form $\max_S |\{(x_1, \ldots, x_k) : \phi\}|$. Then A has a $(1 - 2^{-k_\phi})$-approximation algorithm, where by k_ϕ we denote the number of atomic expressions in ϕ that involve S.

Proof: Consider an instance of A with universe V. For each k-tuple $v = (v_1, \ldots, v_k) \in V^k$ let us substitute these values for $x_1, \ldots, x_k$ in ϕ, to obtain an expression ϕ_v. There are three kinds of atomic expressions of ϕ_v, namely those that have relational symbols G_i (the input relations), $=$ (equality between the v_i's), and S. The former two kinds can be readily evaluated to **true** or **false** from the known values of the input relations and the v_i's, and substituted away from ϕ_v (this should be reminiscent of the similar construction in the proof of Theorem 5.9). It follows that ϕ_v is ultimately a Boolean combination of atomic expressions $S(v_{i_1}, \ldots, v_{i_r})$.

Thus, this instance of A is essentially a set of expressions of the form ϕ_v for all possible k-tuples v, and we are asked to assigning truth values to the various atomic expressions $S(v_{i_1}, \ldots, v_{i_r})$ (which we can consider as Boolean variables) so as to maximize the number of satisfied ϕ_v's. But this is an instance of MAXGSAT (the problem in which we are given a set of Boolean expressions, and we are asked to find the truth assignment that satisfies as many as possible, recall the Maximum Satisfiability subsection of Section 13.1). And the discussion preceding Theorem 13.2 shows how to approximate that problem with relative error at most $(1 - 2^{-m})$, where m is the number of Boolean variables appearing in each expression—in our case, k_ϕ. □

Thus, all optimization problems in **MAXSNP** share a positive approximability property (they all have some ϵ-approximation algorithm with $\epsilon < 1$, though not quite a polynomial-time approximation scheme), much the same way that all problems in **NP** share a positive algorithmic property —they can be solved in polynomial time by a nondeterministic algorithm, though not necessarily by a deterministic one. Whether all problems in **MAXSNP** have a polynomial-time approximation scheme is thus a most important question— the approximability counterpart of the $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ problem that we have been seeking. Predictably, we shall now turn to identifying *complete problems*.

MAXSNP-Completeness

We say that a problem in **MAXSNP** is **MAXSNP**-complete if all problems in **MAXSNP** L-reduce to it. From the Corollary to Proposition 13.2 we have:

Proposition 13.3: If a **MAXSNP**-complete problem has a polynomial-time approximation scheme, then all problems in **MAXSNP** have one. □

Naturally, it is not *a priori* clear that **MAXSNP**-complete problems exist. But they do:

Theorem 13.9: MAX3SAT is **MAXSNP**-complete.

Proof: Since any problem in **MAXSNP** by defintion can be L-reduced to a problem in $\mathbf{MAXSNP}_0$, it suffices to show that all problems in $\mathbf{MAXSNP}_0$ can be L-reduced to MAX3SAT. Consider such a problem A defined by the expression $\max_S |\{(x_1, \ldots, x_k) : \phi\}|$. The proof of Theorem 13.8 essentially shows that A can be L-reduced to MAXGSAT. What we need to do is to further work on the Boolean expressions produced in that construction, so that we obtain an instance of MAX3SAT.

The expressions produced in the proof of Theorem 13.8 for each instance x of A are Boolean expressions of the form ϕ_v, with Boolean variables corresponding to whether the various tuples of constants of x belong to S. As usual, we can discard those expressions ϕ_v that are unsatisfiable. We can represent each of the remaining Boolean expressions ϕ_v as a Boolean circuit with $\wedge$, $\vee$, and $\neg$ gates. We can then use the construction employed in the reduction from CIRCUIT SAT to 3SAT (Example 8.3). That is, we replace each gate of the circuit by a set of two or three clauses stating that the relation between the input and output values are as specified by the sort of the gate (recall Example 8.3). We also add the clause (g), where g is the output gate. We repeat this for each satisfiable ϕ_v. The resulting set of all clauses thus created is the desired instance $R(x)$ of MAX3SAT. From any truth assignment T for this instance we immediately get a feasible solution $S = S(T)$ of the instance x of A, by simply recovering S from the Boolean values of the variables. The description of the L-reduction is complete.

But it remains to show that this is indeed an L-reduction. Each satisfiable expression ϕ_v is replaced by at most c_1 clauses, where c_1 is a constant depending on the size of ϕ, and therefore specific to A (it is essentially three times the number of Boolean connectives in ϕ). Thus, the m satisfiable expressions ϕ_v are replaced by at most $c_1 m$ clauses. The optimum value of instance x is at least some constant fraction of m, say $\text{OPT}(x) \geq c_2 m$ (for example, by Theorem 13.8, we can take $c_2 = 2^{-k_\phi}$). Since in $R(x)$ we can always set the Boolean variables so that all clauses except for those corresponding to the output gates are satisfied, $\text{OPT}(R(x)) \leq (c_1 - 1)m$, and the first condition on the definition of the L-reduction is satisfied with $\alpha = \frac{(c_1-1)}{c_2}$. It is easy to see that the second condition is satisfied, as usual, with $\beta = 1$, and the proof is complete. □

Amplifiers and Expanders

In order to reduce MAX3SAT to other important problems in **MAXSNP** (see Theorem 13.11 below), we need to establish the equivalent of Proposition 9.3, that is, that MAX3SAT remains **MAXSNP**-complete even if each variable appears at most three times in the clauses (we call this restricted problem 3-OCCURRENCE MAX3SAT). In Chapter 9 we proved this by replacing each occurrence of variable x by a new variable, say using variables $x_1, x_2, \ldots, x_k$, and then adding the clauses $(x_1 \Rightarrow x_2), (x_2 \Rightarrow x_3), \ldots, (x_k \Rightarrow x_1)$. This "cycle of implications" ensures that in any satisfying truth assignment all these new variables will have the same truth value.

This simple trick does not work in the present context. The "cycle of implications" construction *is not* an L-reduction from MAX3SAT to 3-OCCURRENCE-MAX3SAT. To see why, let us consider the (admittedly, somewhat far-fetched) instance y of MAX3SAT with clauses $(x), (x), \ldots, (x)$, $(\neg x), (\neg x), \ldots, (\neg x)$, where there are ℓ copies of (x) and ℓ of $(\neg x)$ (recall that we do not forbid fewer than three literals in a clause, or repetitions of clauses). Obviously, the optimum of y is $\text{OPT}(y) = \ell$.

If we perform the reduction sketched above and replace the 2ℓ occurrences of x by $x_1, \ldots, x_{2\ell}$ adding the clauses $(x_1 \Rightarrow x_2), (x_2 \Rightarrow x_3), \ldots, (x_{2\ell} \Rightarrow x_1)$, we obtain a new instance of MAX3SAT $R(y)$ with 4ℓ clauses. We would have liked to say that from any solution s of $R(y)$ we can recover a corresponding solution $S(s)$ of y. However, the optimum solution s of $R(y)$ satisfies all but one of the 4ℓ clauses as follows: $x_1, x_2, \ldots, x_\ell$ are **true**, and $x_{\ell+1}, x_{\ell+2}, \ldots, x_{2\ell}$ are **false**. Thus all clauses $(x_1), \ldots, (\neg x_{2\ell})$ are satisfied; furthermore all clauses of the form $(x_i \Rightarrow x_{i+1})$ are satisfied, *except for the clause* $(x_\ell \Rightarrow x_{\ell+1})$ —the only one with **true** assumption and **false** conclusion. Obviously, there is no meaningful solution $S(s)$ of y that we can recover from s.

Thus, our "cycle of implications" trick does not give us an L-reduction. Is there another, more sophisticated graph of implications that would do? Let us

start by identifying what is wrong with the cycle: There is a large subset of its nodes (namely, $x_1, x_2, \ldots, x_\ell$) that have a single edge leaving them. This is damaging, because it translates in the "cheating" truth assignment that satisfies all clauses except for the implication associated with this edge. What would have prevented such cheating is a directed graph where all sets of nodes have a "substantial" number of edges leaving them—equal to the number of nodes in the set. We formalize this below:

Definition 13.6: Consider a finite set X, and a directed graph $G = (V, E)$ where $X \subseteq V$. We assume that $|V| \leq c \cdot |X|$ where c is a constant, and that all nodes of G have either indegree one and outdegree two or vice-versa, except for nodes in X that have indegree and outdegree one. That is, $|E| \leq 2c \cdot |X|$.

We say that G is an *amplifier* for X if the following is true: For any subset $S \subseteq V$ containing $s \leq \frac{|X|}{2}$ nodes from X, $|E \cap S \times (V - S)|, |E \cap (V - S) \times S| \geq |S|$. That is, there are edges entering and leaving S that are no less than the number of X-nodes in S. □

Suppose that an amplifier G exists for any set X. Then we could use it to build a generalized "cycle of implications," and L-reduce MAX3SAT to 3-OCCURRENCE MAX3SAT, as follows: For each variable of the original formula, say with k occurrences, we connect the set of variables $X = \{x_1, \ldots, x_k\}$, together with up to ck new variables that make up all of V, by no more than $2ck$ implications *exactly as suggested by the edges of* G. We claim that the resulting construction is an L-reduction from MAX3SAT to 3-OCCURRENCE MAX3SAT.

In proof, notice first of all that if y is the original instance, with m clauses, then $R(y)$ has at most $(2c+1)m$ clauses, and thus the first condition is satisfied $\alpha = 2(2c+1)$ (we can satisfy at least half the clauses of y). More importantly, if s is any solution of $R(y)$, then we would be able to modify it so that, for each variable x of y, all k copies of x in $R(y)$, plus all additional nodes in V, have the same truth value—that of the majority of copies of x. This may result in $|S|$ copies of x changing values, and hence in our losing up to $|S|$ clauses satisfied in $R(y)$. However, because of the property of the amplifier, we know that we have *gained* at least $|S|$ clauses in the process (those implications that were going from the **true** to the **false** part of G). Hence, it is no loss of generality to assume that, in the optimum truth assignment of $R(y)$, all copies of each variable have the same truth value, and thus an equivalent truth assignment of y can be readily recovered. The distance from optimality is obviously the same in the two assignments.

We conclude that, if we assume that amplifiers exist, we have a proof that 3-OCCURRENCE MAX3SAT is **MAXSNP**-complete. We shall next show that amplifiers do exist. Once more, our argument will be non-constructive, in a way reminiscent of the proof of Theorem 11.6. We start by defining a related concept:

Definition 13.7: Let $\delta > 0$. A *δ-expander* for X is a graph $G = (X, E)$ where $|E| \leq c'|X|$ with the following property: For each subset $S \subseteq X$ of size s, $|E \cap S \times (V - S)| \geq \delta|S|$. □

From a δ-expander one can construct an amplifier. There are several problems to be fixed. First, there are only $\delta|S|$ edges leaving S instead of the required $|S|$; but this can be achieved by taking $\lceil \frac{1}{\delta} \rceil$ *copies of* G. Second, there is no guarantee that there are many edges *entering* $|S|$; this is easily taken care of *by adding all inverse edges, thus making the graph symmetric.* Finally, to enforce the indegree and outdegree constraints, for each node $x \in X$ with outdegree $k \geq 2$ we add to V $k - 1$ new nodes, that "fan out" the edges leaving x. A similar construction works for the indegrees. The resulting graph is an amplifier for X with $c = \frac{4c'}{\delta}$.

Thus, the following result establishes that amplifiers exist:

Lemma 13.2: For any $n \geq 2$, all sets X of size n have a δ-expander for some $\delta > 0$.

Proof: Consider a *random function F on X*, a random graph with all outdegrees one. That is, for each $x \in X$ we pick equiprobably and independently $F(x)$ to be any other node. Let S be a subset of X of size $|S| = s \leq \frac{n}{2}$. We call S *bad* if it has fewer than δs outgoing edges (where δ is a small number to be fixed later). What is the probability that S is bad?

To answer, delete for a moment from S the sources of the outgoing edges. The remaining subset T of size $|T| \geq (1 - \delta)|S|$ must be *introverted*, that is, for all nodes $x \in T$ we have $F(x) \in S$. For each node of T, the probability that it is mapped inside S is $\frac{s}{n}$, and thus the probability of T being introverted is at most $(\frac{s}{n})^{(1-\delta)|S|}$. The probability that an introverted subset T of S exists is therefore no more than this quantity, multiplied by the number of subsets of S of cardinality $(1 - \delta)s$, which is $\binom{s}{\delta s}$. Since we can approximate $\binom{n}{k}$ from above by $(\frac{en}{k})^k$ (see Problem 13.4.10), this latter number is at most $(\frac{e}{\delta})^{\delta s}$. We conclude that the probability that a set S of cardinality s is bad is no more than $(\frac{e}{\delta})^{\delta s}(\frac{s}{n})^{(1-\delta)s}$.

Suppose now that we take a graph F_k that is the union of k independently chosen random functions on X. Clearly, the probability that S is bad *now* is bounded from above by the kth power of this expression, $(\frac{e}{\delta})^{\delta ks}(\frac{s}{n})^{(1-\delta)ks}$. Finally, since there are at most $(\frac{en}{s})^s$ subsets of X of size s, the expected number of all bad subsets of size s in F_k is at most $(\frac{en}{s})(\frac{e}{\delta})^{\delta ks}(\frac{s}{n})^{(1-\delta)ks}$. Choosing $\delta = \frac{1}{10}$, $k = 8$, and recalling that $s \leq \frac{n}{2}$, after a calculation we conclude that the expected number of s-subsets of X with fewer than $\frac{s}{10}$ outgoing edges in F_8 is at most $(\frac{1}{2})^s$. Adding over all s's, we observe that the expected number of bad subsets in F_8 is *less than one.*

Thus, if we independently choose eight random functions on X and form F_8, the expected number of bad subsets is less than one. Clearly, there must

be one such experiment for which the outcome is zero bad subsets (otherwise the expectation would be at least one). We must conclude that *there is a graph with n nodes and outdegree eight such that all subsets S of size less than $\frac{n}{2}$ have at least $\frac{|S|}{10}$ outgoing edges.* This graph is a δ-expander with $\delta = \frac{1}{10}$ and $c' = 8$. □

Unfortunately, this result still does not give us the sought L-reduction from MAX3SAT to 3-OCCURRENCE MAX3SAT. The reason is that, although we know that amplifiers exist for all n and thus the construction is possible, we have not described an *algorithm* for generating an amplifier in space $\log n$ (in space $n \log n$, of course, we can try all possible graphs until we find our amplifier). The constructive form of Lemma 13.2 is much too involved to prove here (see the references in 13.4.11):

Lemma 13.2′: There is an algorithm which, given n in unary, generates in $\log n$ space an amplifier for a set of size n. □

From this important fact, our main result follows:

Theorem 13.10: 3-OCCURRENCE MAX3SAT is **MAXSNP**-complete. □

From Theorem 13.10 we can show several other **MAXSNP**-completeness results.

Theorem 13.11: The following problems are **MAXSNP**-complete:

(a) 4-DEGREE INDEPENDENT SET;
(b) 4-DEGREE NODE COVER;
(c) 5-OCCURRENCE MAX2SAT;
(d) MAX NAESAT;
(e) MAX-CUT.

Proof: For (a) we notice that the reduction in the proof of Theorem 9.4 is also an L-reduction from 3-OCCURRENCE MAX3SAT to 4-DEGREE INDEPENDENT SET. The properties of L-reduction are obvious, because the optimum value in the two instances is the same. That the maximum degree of the resulting graph is four follows from the fact that, since each variable has at most three occurrences, each literal occurrence has at most two contradicting literal occurrences. The only possible subtlety is that there may now be clauses with one or two literals; these are represented by single nodes or edges, respectively (as opposed to triangles).

Reducing 4-DEGREE INDEPENDENT SET to 4-DEGREE NODE COVER is trivial: The R part is the identity, and the S part produces $V - S$ from S (recall the Corollary 2 to Theorem 9.4). The reduction is an L-reduction (despite the fact that the same reduction between the unrestricted problems *is not* an L-reduction, recall Example 13.1), because both optimum values are lineraly related to the number of nodes.

The L-reduction from 4-DEGREE INDEPENDENT SET to 5-OCCURRENCE MAX2SAT, we have a variable for each node. For each node x we add to the instance of 5-OCCURRENCE MAX2SAT the clause (x), and for each edge $[x, y]$ we add the clause $(\neg x \vee \neg y)$. This completes the construction of the instance of 5-OCCURRENCE MAX2SAT. It is easy to see that the optimum truth assignment can be assumed to satisfy all edge clauses (because, if not, making one of the nodes **false** cannot decrease the number of satisfied clauses). Thus the optimum number of satisfied clauses equals the size of the maximum independent set plus $|E|$—and $|E|$ is a constant multiple of the maximum independent set, recall Example 13.1.

We can reduce MAX2SAT to MAX NAESAT (the maximization problem associated with the not-all-equal version of 3SAT) by the same reduction we used to show NAESAT **NP**-complete (Theorem 9.3): We add to each clause a new literal z. The optimum value in the two instances is the same, and so we have an L-reduction. Finally, the reduction from NAESAT to MAX-CUT in the proof of Theorem 9.5 is an L-reduction. □

13.3 NONAPPROXIMABILITY

We have been treating the question of whether **MAXSNP**-complete problems have polynomial-time approximation schemes very much like the $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ question. As it happens, the similarity runs a little deeper than anyone had expected: A sequence of deep and striking recent results has established that *the two questions are equivalent:* **MAXSNP**-complete problems have polynomial-time approximation schemes if and only if $\mathbf{P} = \mathbf{NP}$—naturally, this is the strongest negative result we could have hoped for approximability, short of proving that $\mathbf{P} \neq \mathbf{NP}$. Connecting the approximability issue with the $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ question required the development of a very interesting *alternative characterization of* **NP**, one that sees this class in a light much more appropriate for the discussion of the issue of approximability than our current view in terms of precise, rigid computations. We describe this fascinating result next.

Weak Verifiers

Let L be a language and M a Turing machine. We say that M *is a verifier for* L if L can be written as

$$L = \{x : (x, y) \in R \text{ for some } y\},$$

where R is a polynomially balanced relation decided by M. According to Proposition 9.1, a language L is in **NP** if and only if it has a polynomial-time verifier. As it turns out, the requirement that M be polynomial time can be relaxed:

Proposition 13.4 (Cook's Theorem, Weak Verifier Version): A language L is in **NP** if and only if it has a deterministic logarithmic-space verifier.

Proof: Since SAT is **NP**-complete, there is a reduction F from L to SAT. Define now $(x, y) \in R$ if and only if $y = F(x); z$, where z is a satisfying truth assignment of the Boolean expression $F(x)$. Obviously $x \in L$ if and only if there is a y such that $(x, y) \in R$. Furthermore, whether $(x, y) \in R$ can be decided in logarithmic space: The machine computing $F(x)$ compares one-by-one the bits of its output with y; telling whether z satisfies $F(x)$ in logarithmic space is trivial. □

So, the complexity of verifiers for **NP** can be reduced from polynomial-time to logarithmic-space. How much further can it be reduced? The important result of this subsection is a *surprisingly weak verifier for* **NP**.

Definition 13.8: Our new verifier is a *randomized* machine which makes use of *only* $\mathcal{O}(\log |x|)$ *random bits* when verifying whether $(x, y) \in R$. Furthermore, although the verifier has full access to the input x, *it has very limited access to the certificate* y*: It only examines a constant number of bits of* y. This is done as follows: On input x, and with random bit string $r \in \{0,1\}^{\lceil c \log |x| \rceil}$ for some $c > 0$, the verifier computes a *finite set of integers* $Q(x, r) = \{i_1, \ldots, i_k\}$, where k is a fixed integer and the i_j's are all at most $|y| = p(|x|)$. Then the i_jth symbol of y is found and written on a string of the verifier, $j = 1, \ldots, k$; the rest of the certificate y is not needed. Finally, the verifier performs a polynomial-time computation based on x, r, and $y_{i_1} \ldots y_{i_k}$. All computations of the verifier end with a "yes" or a "no".

We call such a machine a $(\log n, 1)$-*restricted verifier*, to record its two important resource restrictions: $\mathcal{O}(\log n)$ random bits, and a constant, $\mathcal{O}(1)$ that is, number of access steps. We say that a $(\log n, 1)$-restricted verifier decides a relation R if for each input x and alleged certificate y the following is true: If $(x, y) \in R$ then for all random strings the verifier ends up with "yes". If however $(x, y) \notin R$, then at least $\frac{1}{2}$ of the random strings cause the verifier to end up with "no". In other words, there can be a limited fraction of false positives, but no false negatives†. □

The novel characterization of **NP** alluded to above is this:

Theorem 13.12: A language L is in **NP** if and only if it has a $(\log n, 1)$-restricted verifier. □

---

† Recall that our goal here is to provide an alternative view of **NP**, one that will rid us of the "rigidity" of the conventional view, and thus create a framework appropriate for the study of approximability. In this regard randomness seems a most important ingredient, in that the possibility of false positives provides the necessary "imprecision" that was lacking in the conventional formulation of **NP**. This is reminiscent of the definition of $\mathbf{MAXSNP}_0$ (Definition 13.5), where instead of rigidly insisting that ϕ hold for all k-tuples, we settled for the set S that maximizes the number of k-tuples for which ϕ holds. It can be shown that the two maneuvers are in fact equivalent (see Problem 13.4.13).

The ingenious proof of Theorem 13.12 brings together many ideas and techniques that have been developed in recent years. For an account see 20.2.16 and 20.2.17.

Nonapproximability Results

Theorem 13.12 has some immediate and important consequences for the approximability of optimization problems:

Theorem 13.13: If there is a polynomial-time approximation scheme for MAXSAT then $\mathbf{P} = \mathbf{NP}$.

Proof: Let $L \in \mathbf{NP}$, and let V be the $(\log n, 1)$-restricted verifier that decides the relation R as in Theorem 13.12, using $c \log n$ random bits and d accesses, for some positive constants c and d. It is not a loss of generality to assume that if $(x, y) \in R$ then $x; y \in \{0,1\}^*$ and $|x; y| = |x|^k$. Suppose now that there is a polynomial-time approximation scheme for MAXSAT which achieves approximation $\epsilon > 0$ in time $p_\epsilon(n)$, a polynomial. Based on this, we shall describe a polynomial-time algorithm for deciding L.

Suppose that we wish to tell whether $x \in L$, where $|x| = n$. Let $r \in \{0,1\}^{c \log n}$, and consider the computation of V effected by this sequence of random choices. We wish to construct a Boolean expression that expresses the fact that the computation ends up in "yes". During this computation, V will seek access to d bits of y, say $y_{i_1(r)}, \ldots, y_{i_d(r)}$. Except for these bits of y, all other aspects of the computation of V or random choices r are completely determined. Thus, the outcome of the computation is a *Boolean function of these d bits considered as Boolean variables*, and thus can be expressed as a circuit C_r. We know (recall the proof of Proposition 4.3) that the number of gates of C_r is at most $K = 2^{2d}$, a constant. In turn, C_r can be expressed as a set of K or fewer clauses, denoted ϕ_r (recall the reduction from CIRCUIT SAT to SAT, Example 8.3). Notice that, no matter how we set the values of the variables $y_{i_1(r)}, \ldots, y_{i_d(r)}$, all but one of the clauses can be satisfied. Only certain settings of these variables can satisfy the last clause (the one stating that the outcome of the computation is "yes").

Repeating for all $2^{c \log n} = n^c$ possible sequences of random choices for V, we arrive at a set of at most Kn^c clauses, where the various groups of clauses share only the $y_{i_j(r)}$ variables. Because of the acceptance convention of our verifiers, we know the following: If $x \in L$, then there is a truth assignment (that is, a certificate y for x, together with the values of the gates of the various circuits C_r) that satisfies all clauses. And if $x \notin L$, any truth assignment must miss one clause in at least half of the groups; that is, at least a fraction of $\frac{1}{2K}$ of the clauses must be left unsatisfied.

This is where the assumed polynomial-time approximation scheme for MAXSAT comes into play. We apply this scheme to the constructed set of clauses

with $\epsilon = \frac{1}{4K}$ -the time requirements will be $p_{\frac{1}{4K}}(Kn^c)$, a polynomial in n. If the scheme returns a truth assignment that satisfies more than a fraction of $1 - \frac{1}{2K}$ of the clauses, then we know that $x \in L$ (otherwise there is no truth assignment that satisfies more than such a fraction of the clauses). Otherwise, since the returned truth assignment is guaranteed to be within $\frac{1}{4K}$ of the optimum, we know that there is no truth assignment that satisfies all clauses, and thus $x \notin L$ (notice the similarity with the argument in the proof of Theorem 13.4 about the nonapproximability of the TSP). The proof is complete. □

Since MAXSAT is in **MAXSNP**, we must conclude that *no* **MAXSNP**-*complete problem can have a polynomial-time approximation scheme*, unless of course **P** = **NP**:

Corollary 1: Unless **P** = **NP**, none of these problems have a polynomial-time approximation scheme: MAX3SAT, MAXNAESAT, MAX2SAT, 4-DEGREE INDEPENDENT SET, NODE COVER, and MAX-CUT. □

See the references for more results of this sort. For the unrestricted INDEPENDENT SET problem, we combine Theorem 13.13 with Theorem 13.6 to get something much more devastating:

Corollary 2: Unless **P** = **NP**, the approximation threshold of INDEPENDENT SET and CLIQUE is *one*. □

13.4 NOTES, REFERENCES, AND PROBLEMS

13.4.1 Problem: (a) Show that the greedy heuristic for NODE COVER (recall Section 13.1) never produces a solution which is more than $\ln n$ times the optimum.

(b) Find a family of graphs in which the $\ln n$ bound is achieved in the limit.

(c) NODE COVER is a special case of SET COVERING (recall Corollary 1 to Theorem 9.9). Why? Generalize the greedy heuristic to SET COVERING, and show that it has the same worst-case ratio.

These results are from

- D. S. Johnson "Approximation algorithms for combinatorial problems," *J.CSS*, *9*, pp. 256–278, 1974,

which was the first systematic work on the subject of approximability of **NP**-complete problems. For the generalization in (c) see

- V. Chvátal "A greedy heuristic for the set cover problem," *Math. of Operations Research*, *4*, pp. 233–235, 1979.

It has been shown that the approximability of SET COVERING is one (and in fact, unless $\mathbf{P} = \mathbf{NP}$, the best possible ratio is $\Theta(\log n)$), see

- C. Lund and M. Yannakakis "On the hardness of approximating minimization problems," *Proc. 25th ACM Symp. on the Theory of Computing*, pp. 286 295, 1993.

13.4.2 Problem: Recall the formal discussion of pseudopolynomial algorithms in Problem 9.5.31. Suppose that a strongly **NP**-complete optimization problem has the property that, for all inputs x, the optimum cost is at most $p(\text{NUM}(x))$ for some polynomial $p(n)$. (Notice that all problems we have seen so far in this book satisfy this property, with the intended meaning of $\text{NUM}(x)$.)

Show that such a problem has a fully polynomial-time approximation scheme if and only if $\mathbf{P} = \mathbf{NP}$.

13.4.3 Problem: Recall the BIN PACKING problem (Theorem 9.11); its minimization version seeks the smallest number of bins possible. Show that the approximation threshold of BIN PACKING is at least $\frac{1}{3}$. (Consider the problem of telling whether the number of bins needed is two or three, and recall the PARTITION problem shown **NP**-complete in Problem 9.5.33.)

13.4.4 Asymptotic approximation. There is something unsatisfying about the negative approximability result in the previous problem: It only holds for very small values of the number of bins. For all we know, there may be a polynomial-time heuristic for BIN PACKING that always comes within one bin of the optimum!

Obviously, this calls for a definition: We say that a heuristic M is an *asymptotic ϵ-approximation algorithm* if there is a constant $\delta > 0$ such that for all instances x

$$|c(M(x)) - \text{OPT}(x)| \leq \epsilon \cdot \max\{\text{OPT}(x), c(M(x))\} + \delta.$$

The *asymptotic approximation threshold* is again the greater lower bound for all ϵ's for which there is an asymptotic ϵ-approximation algorithm. If the asymptotic approximation threshold is zero, we say that the problem has an asymptotic polynomial-time approximation scheme.

(a) Consider an optimization problem A (maximization with goal K, or minimization with budget B), and the special case of problem A in which the goal (or budget) is bounded above by a constant c. We call this special case the *constant restriction* of A. Which of these problems have polynomial-time solvable constant restrictions, and which have an **NP**-complete constant restriction?

TSP, MINIMUM COLORING, MAX-CUT, BIN PACKING, KNAPSACK.

(b) Prove: If all constant restrictions of a problem are polynomial-time solvable, then its asymptotic approximation threshold coincides with the ordinary approximation threshold.

13.4.5 Problem: Problem 13.4.3 establishes that the approximation threshold for BIN PACKING is at least $\frac{1}{3}$, but tells us nothing about its asymptotic approximation threshold, arguably the more interesting quantity.

Consider then the following "first-fit" heuristic, where the n items are $a_1, \ldots, a_n$ and the capacity is C:

> Initialize n bins to empty $B[j] := 0, j = 1, \ldots, n$.
> For $i = 1, \ldots, n$ do:
> Find the smallest j with $B[j] + a_i \leq C$, and set $B[j] := B[j] + a_i$

(a) Show that the first-fit heuristic leaves at most one bin less than half-full.

(b) Conclude that the asymptotic approximation threshold of BIN PACKING is at most $\frac{1}{2}$.

13.4.6 Problem: But we can do much better; BIN PACKING is an example of a strongly **NP**-complete problem that has a polynomial-time asymptotic approximation scheme.

Fix any $\epsilon > 0$, and let $Q = \lfloor \epsilon C \rfloor$ be the "quantum size." Replace each item a_i by the quantity $a'_i = \lceil \frac{a_i}{Q} \rceil$. Notice that the value of each item must now be one of the "standardized" values $\{1, 2, \ldots, k\}$, where $k = \mathcal{O}(\frac{1}{\epsilon})$.

A *pattern* is a *sorted* sequence of positive integers adding up to k or less. For example, if $k = 4$, then $(1, 1, 2)$ and (3) are patterns. For $k = 4$ there are 12 different patterns: $\{(), (1), (1, 1), (1, 1, 1), (1, 1, 1, 1), (1, 1, 2), (1, 2), (1, 3), (2), (2, 2), (3), (4)\}$. In general, the number of patterns will be a fixed number P depending (exponentially) on k.

(a) Express the requirement that the items a'_i must be packed in m bins of capacity k as a set of equations in the nonnegative integer variables $x_1, \ldots, x_P$, where the intended meaning of x_j is *how many of the m bins are filled according to the jth pattern.*

(b) Using the fact that INTEGER PROGRAMMING with a constant number of variables can be solved in polynomial time (recall the discussion in 9.5.34), prove that there is an asymptotic polynomial-time approximation scheme for BIN PACKING.

For even better approximation algorithms for BIN PACKING see

- N. Karmarkar and R. M. Karp "An efficient approximation scheme for the one-dimensional bin-packing problem," *Proc. 23rd IEEE Symp. on the Foundations of Computer Science*, pp. 312–320, 1982.

13.4.7 Problem: Show that the MINIMUM UNDIRECTED KERNEL problem (recall Problem 9.5.10(g)) has approximation threshold one. (Modify the reduction used in its **NP**-completeness proof.)

13.4.8 Problem: Suppose that in a TSP instance the distances satisfy the *triangle inequality* $d_{ik} \leq d_{ij} + d_{jk}$. An approximation algorithm for this special case of the TSP is based on this idea: We find the *minimum spanning tree* of the cities (recall Problem 9.5.13). Taking each edge of the tree twice we get a graph that is connected, and has all degrees even. Such graphs are called *Eulerian*.

(a) Show that an Eulerian graph (possibly with multiple edges) has a cycle that visits each edge once (and each node possibly more than once).

(b) Show that in an instance of the TSP, if we have an Eulerial graph with total cost K, then we can find a tour with cost K or better.

(c) Show that there is a polynomial-time heuristic for the TSP with triangle inequality that yields a solution at most *twice* the optimum. (How does the minimum spanning tree compare with the optimum tour?)

There is a more sophisticated $\frac{2}{3}$-approximate heuristic, based on a *minimum matching of the odd-degree nodes* of the minimum spanning tree:

- N. Christofides "Worst-case analysis of a new heuristic for the traveling salesman problem," technical report GSIA, Carnegie-Mellon Univ., 1976.

This is the best heuristic known for the TSP with triangle inequality.

Suppose now that all distances in a TSP instance are either one or two. We wish to prove that this special case of the TSP is **MAXSNP**-complete. A first difficulty is this: Why is this special case of the TSP in **MAXSNP**?

It turns out that *any optimization problem with approximation threshold strictly less than one is in* **MAXSNP***!* In other words, **MAXSNP** is precisely the class of all optimization problems with approximation threshold strictly less than one (this is a result due to Madhu Sudan and Umesh Vazirani, communicated to me in 1993). To exemplify this for the TSP with distances one and two, we shall L-reduce it to MAXSAT. Given such an instance of the TSP with n cities, we know that its optimum value is between n and $2n$. Since telling whether the optimum of such an instance of the TSP is at most a given integer k is a problem in **NP**, for each k between n and $2n$ we can produce a Boolean expression ϕ_k such that (1) if the optimum is at most k c_k clauses of ϕ_k can be satisfied; and (2) otherwise all $c_k + 1$ clauses are satisfied.

Thus, the optimum of the instance of MAXSAT that combines all these clauses is $\sum_{k=n}^{2n} c_k + t$, where t is the optimum tour length.

(d) Show that this special case of the TSP is **MAXSNP**-complete.

When all distances are 1 or 2, the triangle inequality still holds (why?) and thus Christofides's heuristic still is still $\frac{1}{3}$-approximate. However, for the $1-2$ special case there is a polynomial $\frac{1}{6}$-approximate algorithm (and this is the best known):

- C. H. Papadimitriou and M. Yannakakis, "The traveling salesman problem with distances one and two," *Math. of Operations Research, 18*, 1, pp. 1–12, 1993.

Part (d) is also proved there.

Incidentally, for the *asymmetric* generalization of the TSP, where d_{ij} is not necessarily equal to d_{ji}, but the triangle inequality still holds, no polynomial heuristic achieving a constant ratio is known.

13.4.9 Problem: (a) Show that, unless **P** = **NP**, the approximation threshold of STEINER TREE is one.

(b) Prove that the STEINER TREE problem is **MAXSNP**-complete even when all distances are 1 and 2.

(c) Consider the following heuristic for the STEINER TREE problem with triangle inequality: First, find the shortest distances between all nodes in the graph. Then, find the minimum spanning tree of the mandatory nodes, using as distances the shortest path lengths just found. Finally, build a Steiner tree by putting together all shortest paths used in the shortest spanning tree. Show that this algorithm never yields a Steiner tree with cost more than twice the optimum; thus the approximation threshold of STEINER TREE with triangle inequality is at most $\frac{1}{2}$.

The heuristic described above is from

- L. Kou, G. Markowsky, and L. Berman "A fast algorithm for Steiner trees," *Acta Informatica, 15*, pp. 141-145, 1981.

More recently, it was shown that the approximation threshold of STEINER TREE with triangle inequality is at most $\frac{5}{11}$:

- A. Z. Zelikowski "An $\frac{11}{6}$-approximation algorithm for the network Steiner problem," *Algorithmica, 9*, 5, pp. 463–470, 1993.

13.4.10 The doctoral dissertation

- V. Kann *On the Approximability of NP-complete Optimization Problems,* Royal Institute of Technology, Stockholm, Sweden, 1991

contains a comprehensive review of approximability results ca. 1991, as well as an extensive list of optimization problems and their approximability status.

13.4.11 Problem: Use Stirling's approximation formula $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ to show that $\binom{n}{k} \leq (\frac{en}{k})^k$.

13.4.12 The latest and simplest proof of Lemma 13.2′ can be found in

- O. Gabber and Z. Galil "Explicit construction of linear-sized superconcentrators," *J.CSS 22*, 407 420, 1981.

13.4.13 Problem: (a) Show that, unless **P** = **NP**, the approximation threshold of MINIMUM COLORING cannot be less than $\frac{1}{4}$. (Just recall that telling whether the minimum number of colors is 3 or 4 is **NP**-complete, Theorem 9.8.)

(b) Show that the *asymptotic* approximation threshold of MINIMUM COLORING cannot be less than $\frac{1}{4}$. (Replace each node with a large clique.)

(c) Can you further amplify the construction for (b) to show that, unless **P** = **NP**, the asymptotic approximation threshold of MINIMUM COLORING cannot be less than $\frac{1}{2}$?

Part (c) was proved in

- M. R. Garey and D. S. Johnson "The complexity of near-optimal graph coloring," *J.ACM, 23*, pp. 43–49, 1976.

It was the strongest negative result known about this important problem until it was proved in the paper by Lund and Yannakakis referenced in 13.4.1 that the approximation threshold for MINIMUM COLORING is *one*.

13.4.14 MAXSNP and weak verifiers. We could have defined the class of optimization problems **MAXSNP** not in terms of logical expressions, but in terms of $(\log n, 1)$-restricted verifiers (recall Theorem 13.12).

Let $k_1, k_2, k_3 > 0$, and let f be a polynomial-time computable function assigning to each string x and each bit string r with $|r| = k_2 \log |x|$ a set $f(x, r)$ of k_1 numbers in the range $1 \ldots |x|^{k_3}$, and let M be a polynomial-time Turing machine with three inputs. Define now this optimization problem:

"Given x, find the string y of length $|x|^{k_3}$ that maximizes the number of strings r with $|r| = k_2 \log |x|$ such that $M(x, r, y|_{f(x,r)}) =$ "yes"."

Let **MAXPCP**† be the class of all optimization problems of this form.

(a) Show that **MAXSNP** $\subseteq$ **MAXPCP**. (For each problem in **MAXSNP**$_0$ defined in terms of an expression ϕ, function f computes, for each input relation G and values for the first-order variables, the finitely many positions in relation S that we must look in order to decide ϕ. What if the problem is not in **MAXSNP**$_0$?)

(b) Show that **MAXPCP** $\subseteq$ **MAXSNP**. (The trick is to define the right input relation G for each f, M, and input x. G has $K = k_1 + k_2 + k_1 \cdot k_3$ arguments; we write $G(r_1, \ldots, r_{k_2}, b_1, \ldots, b_{k_1}, j_{11}, \ldots, j_{1k_3}, \ldots, j_{k_1 k_3})$. A K-tuple of integers in the range $0 \ldots |x| - 1$ are related by G if the following is true: (a) The ith element of $f(x, r)$ is

---

† **PCP** stands for "probabilistically checkable proofs." **PCP**$(\log n, 1)$ was the name given in the paper referenced in 13.4.15 below to the class of all languages that have $(\log n, 1)$-restricted verifiers—that is to say, the class **NP**. The purpose of this problem is to point out the close relationship between this concept and **MAXSNP**.

the integer $j_{i1} \ldots j_{ik_3}$ in $|x|$-ary, where r is the bit string spelled by the r_j's in binary; and (b) If the bit of y that corresponds to the ith element of $f(x,r)$ equals b_i for all i, then $M(x,r,y|_{f(x,r)}) =$ "yes". On the other hand, relation S encodes y (say, it contains all k_3-tuples that correspond to 1-bits of y). Write a simple expression ϕ that says "$M(x,r,y|_{f(x,r)}) =$ "yes"".

13.4.15 The definition of **MAXSNP** and Theorems 13.8 through 13.11 are from

- C. H. Papadimitriou and M. Yannakakis, "Optimization, approximation, and complexity classes," *Proc. 20th ACM Symp. on the Theory of Computing*, pp. 229–234, 1988; also, *J.CSS* 1991.

13.4.16 Theorems 13.12 and 13.13 were proved in

- S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy "Proof verification and hardness of approximation problems," *Proc. 33rd IEEE Symp. on the Foundations of Computer Science*, pp. 14–23, 1992.

The proof of Theorem 13.12 is the culmination of several lines of investigation in complexity theory. Most immediately, it builds upon a slightly weaker result —namely that **NP** has $(\log n, (\log \log n)^k)$-restricted verifiers, as opposed to $(\log n, 1)$-restricted ones, enough to prove Corollary 2 of Theorem 13.13— which preceded it by a few weeks, and was presented in the same conference

- S. Arora, S. Safra "Probabilistic checking of proofs," *Proc. 33rd IEEE Symp. on the Foundations of Computer Science*, pp. 2-13, 1992.

For an account of the developments that led to this remarkable result see Problems 20.2.16 and 20.2.17, as well as the following edition of the "NP-completeness column."

- D. S. Johnson "Tha tale of the second prover," *J. of Algorithms 13*, pp. 502–524, 1992.

CHAPTER

14 ON P vs NP

Are the results in this chapter our first steps towards resolving the $P \stackrel{?}{=} NP$ question? Was the hot air balloon our first breakthrough in the conquest of space? It is a matter of opinion and perspective.

14.1 THE MAP OF NP

The usefulness of **NP**-completeness for classifying problems in **NP** cannot be overstated. Once in a while, however, one comes across a problem that resists such a classification: After much effort, neither a polynomial-time algorithm nor an **NP**-completeness proof is forthcoming. The GRAPH ISOMORPHISM problem defined in Section 12.2 is an often-mentioned example, but by no means the only one; certain problems in the various "semantic classes" that we have considered cannot be categorized either. The question arises naturally: *Are there problems in* **NP** *that are neither in* **P** *nor* **NP***-complete?*

For all we know **P** could be equal to **NP**, in which case this question would be meaningless: Everything in **NP** would be *both* polynomial-time computable *and* **NP**-complete (this latter holds if we allow for a moment a slightly more generous notion of reduction, *the polynomial-time reduction*, instead of our standard logarithmic-space reduction). This unlikely situation is depicted in Figure 14.1(c). On the other hand, probably $\mathbf{P} \neq \mathbf{NP}$ and the situation looks something like Figure 14.1(a). Or could it be like Figure 14.1(b), with all problems in **NP** neatly classified as either in **P** or **NP**-complete? The next result states that *Figure 14.1(b) is impossible:* The world is either as in (a), or as in (c). And ruling out one out of three isn't bad at all for a start...

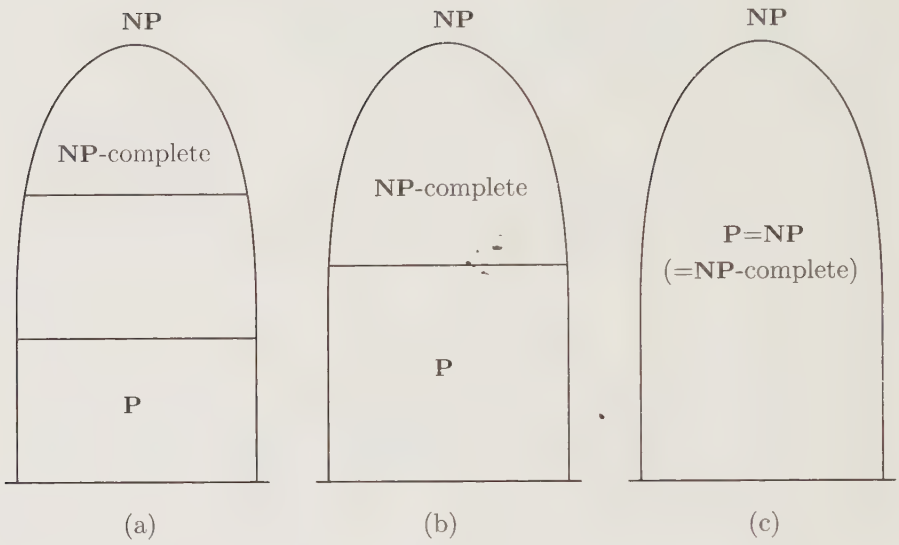


Figure 14-1. Three tentative maps of **NP**.

Theorem 14.1: If $\mathbf{P} \neq \mathbf{NP}$, then there is a language in **NP** which is neither in **P** nor is it **NP**-complete.

Proof: We assume that we have an enumeration $M_1, M_2, \ldots$ of all polynomial-time bounded Turing machines (each with a polynomial "alarm clock"), and an enumeration $R_1, R_2, \ldots$ of all logarithmic-space reductions (equipped with a logarithmic yardstick). Such enumerations can be designed in a variety of easy ways. For example, we can generate systematically all possible Turing machine transition tables, and attach to each of them a standard alarm clock that counts up to n^k, or a logarithmic yardstick. All we need in the proof is that there is a Turing machine that produces the M_i's one after the other; similarly for the R_i's [1]. We also assume that we have a deterministic Turing machine S that decides SAT (presumably in exponential time, since we are assuming that $\mathbf{P} \neq \mathbf{NP}$).

We describe now the language L that is neither **NP**-complete nor in **P**. In fact, we shall describe it in terms of a machine K that decides it. K is simply this:

---

[1] The reader is warned that not all such enumeration problems are so easy. For example, it is not known whether one can enumerate the languages in any of the semantic classes we have considered—in some sense, this is exactly what we mean by "semantic class." For an example where enumeration is possible but nontrivial, see Problem 14.5.2.

$K(x)$: If $S(x) =$ "yes" and $f(|x|)$ is even, then accept x, else reject x. (1)

In other words, a string is accepted by K if and only if it encodes a satisfiable Boolean expression in conjunctive normal form, and function f of its length is an even number, where f is a computable function from integers to integers, whose definition is the heart of the proof.

Now for the definition of f. It is a non-decreasing function, that is $f(n+1) \geq f(n)$, but it grows *very* slowly. We shall describe a Turing machine F that computes f, where the input n is encoded as 1^n on the input string. F operates in two stages, and each stage lasts for just n steps of the machine. We can think that F moves its input cursor to the right at each step, and so it terminates the first stage when it sees the first blank. Then it walks all the way back, and ends the second stage when it sees a $\triangleright$. During the first stage, F starts computing $f(0), f(1), f(2)$, etc., *as many of these as it is able to complete in n steps.* Suppose that the last value of f thus computed was $f(i) = k$. *The value of $f(n)$ will be either k or $k+1$, as determined by the second stage.*

As promised, f already seems to grow very slowly. If $n(k)$ is the smallest number for which $f(n) = k$, it is clear that the smallest number for which $f(n)$ has a chance at becoming $k+1$ is at least $\frac{n(k)^2}{2}$. It follows that $f(n) = \mathcal{O}(\log \log n)$, a very slow-growing function indeed. But, as we shall see, f may end up growing even more slowly...

Now the second stage starts. Exactly what is done in the second stage depends on whether k is odd or even. First, suppose that $k = 2i$ is even. Then F starts simulating the computations $M_i(z)$, $S(z)$, and $F(|z|)$, where z ranges lexicographically over all strings in Σ^* of length $0, 1, 2, \ldots$, and so on, again *as many z's as it is possible to do within n steps of machine F.* What F is trying to find is a string z such that

$$K(z) \neq M_i(z), \tag{2}$$

where K is the machine deciding L, described above. By the definition of K, we are looking for a z such that either (a) $M_i(z) =$ "yes" and either $S(z) =$ "no" or $f(|z|)$ is odd; or (b) $M_i(z) =$ "no", $S(z) =$ "yes", and $f(|z|)$ is even. If such a z is found within the allotted n steps, then $f(n) = k+1$; otherwise $f(n) = k$. This completes the description of the second stage for the case k is even.

Suppose now that $k = 2i - 1$ is odd. Then in its second stage F simulates for n steps the computations $R_i(z)$ (this is a reduction, and thus produces a string), $S(z)$, $S(R_i(z))$, and $F(|R_i(z)|)$, again going patiently over all strings, in lexicographic order. Now F is looking for a z such that

$$K(R_i(z)) \neq S(z). \tag{3}$$

That is, it must be the case that either (a) $S(z) =$ "yes" and either $S(R_i(z)) =$ "no" or $f(|R_i(z)|)$ is odd; or (b) $S(z) =$ "no", $S(R_i(z)) =$ "yes", and $f(|R_i(z)|)$

is even. Again, if such a x is found within n steps then $f(n) = k + 1$; otherwise $f(n) = k$.

F is a well-defined machine, and computes an integer function f, where $f(n)$ can be computed in $\mathcal{O}(n)$ time. Thus, K in (1) is also well-defined, and decides a language L. It is easy to see that $L \in \mathbf{NP}$: On input x we have to do two things: Guess a satisfying truth assignment, and compute $f(|x|)$ to make sure that it is even. Both can be done in nondeterministic polynomial time.

We claim that L is not in $\mathbf{P}$, neither is it $\mathbf{NP}$-complete. Suppose first that $L \in \mathbf{P}$. Then L is accepted by some polynomial machine in our enumeration, say M_i. That is, $L = L(M_i)$, or $K(z) = M_i(z)$ for all z. However, if this is the case, the second stage of F with $k = 2i$ will never find a z for which (2) holds, and so $f(n) = 2i$ for all $n \geq n_0$, for some integer n_0. Consequently, $f(n)$ is even for all but finitely many n; and thus L coincides with SAT on all but finitely many strings. But this contradicts our two assumptions: That $L \in \mathbf{P}$, and $\mathbf{P} \neq \mathbf{NP}$. So, $L \notin \mathbf{P}$.

So, suppose that L is $\mathbf{NP}$-complete. A similar contradiction is near: Since L is $\mathbf{NP}$-complete, there is a reduction, say R_i in our enumeration, from SAT to L. That is, for all z $K(R_i(z)) = S(z)$. It follows that stage two of F will fail to find an appropriate z when $k = 2i - 1$, and thus $f(n) = 2i - 1$ for all but finitely many n. But, recalling the definition of L via K, this implies that L is a finite language. Hence L is in $\mathbf{P}$, absurd since it was assumed that it is $\mathbf{NP}$-complete, and also that $\mathbf{P} \neq \mathbf{NP}$. □

14.2 ISOMORPHISM AND DENSITY

All $\mathbf{NP}$-complete problems are very intimately related, since any one of them can be mapped to any other by a reduction. But there is a stronger and somewhat surprising statement to be made: All *known* $\mathbf{NP}$-complete languages are in fact *polynomially isomorphic*.

Definition 14.1: We say that two languages $K, L \subseteq \Sigma^*$ are *polynomially isomorphic* if there is a function h from Σ^* to itself such that:

(i) h is a bijection, that is, it is one-to-one and onto;
(ii) For each $x \in \Sigma^*$ $x \in K$ if and only if $h(x) \in L$; and
(iii) Both h and its inverse h^{-1} (a bijection has a well-defined total inverse) are polynomial-time computable.

Functions h and h^{-1} are then called *polynomial-time isomorphisms*. □

Example 14.1: Polynomial-time isomorphisms are, strictly speaking, not necessarily reductions, since they use polynomial time, not just logarithmic space, like reductions do. But which reductions are isomorphisms? A reduction that is a bijection would be a good candidate. Unfortunately most reductions fail to be bijections, simply because they are usually not one-to-one, and almost never

onto—recall that all our reductions produce instances that are *very specialized*, and therefore cannot cover all instances. For a rare but somewhat trivial exception, recall the reduction between CLIQUE and INDEPENDENT SET —the one that maps an input (G, K) to $(\overline{G}, K)$, where $\overline{G}$ is the complement of G. Obviously this mapping is one-to-one and onto, polynomial-time computable, and its inverse (which happens to be itself) is also polynomial. □

But in general, designing reductions that are bijections can be very challenging. Fortunately, there is a simple, systematic way to turn reductions into bijections—of course, with no harm to their low complexity. We explain it next. First, there is a simple way to make reductions *one-to-one, length-increasing, and efficiently invertible*. This is based on the idea of *padding functions*.

Definition 14.2: Let $L \subseteq \Sigma^*$ be a language. We say that a function pad : $(\Sigma^*)^2 \mapsto \Sigma^*$ is a *padding function for* L if it has the following properties:

(i) It is computable in logarithmic space.
(ii) For any $x, y \in \Sigma^*$, $\text{pad}(x, y) \in L$ if and only if $x \in L$.
(iii) For any $x, y \in \Sigma^*$, $|\text{pad}(x, y)| > |x| + |y|$.
(iv) There is a logarithmic-space algorithm which, given $\text{pad}(x, y)$ recovers y.

That is, the function pad is essentially a length-increasing reduction from L to itself that "encodes" another string y into the instance of L. □

Example 14.2: Consider SAT. Given an instance x of this problem with n variables and m clauses, and another string y, we can define $\text{pad}(x, y)$ as follows: It is an instance of SAT containing all clauses of x, plus $m + |y|$ more clauses, and $|y| + 1$ more variables. The first m additional clauses are just copies of the clause (x_{n+1}), while the $m + i$th additional clause, for $i = 1, \ldots, |y|$, is either $(\neg x_{n+i+1})$ or (x_{n+i+1}), *depending on whether the ith symbol of* y *was* 0 *or* 1—here we assume with no loss of generality that Σ is $\{0, 1\}$.

We claim that pad is a padding function for SAT. First, it is clearly computable in logarithmic space. Second, it does not affect at all the satisfiability of x, because it just adds disjoint, satisfiable clauses to it. Also, it is clearly length-increasing. Finally, given $\text{pad}(x, y)$ we can identify easily where the extraneous part starts (this is why we have used so many copies of a clause; other tricks would do here), and then recover y from the remaining clauses (decoding $(\neg x_{n+i+1})$ as a 0 and (x_{n+i+1}) as a 1). □

Example 14.3: Consider the CLIQUE language: Given a graph $G = (V, E)$ and an integer K, is there a clique of size K? We can assume that G is connected and $K > 2$.

Here is a simple padding function for CLIQUE. $\text{pad}(G, K, y)$ retains the same K, and attaches to G (say, from node 1) a long tree of nodes (see Figure 14.2). The tree starts off as a long path of $|V|$ nodes—again, in order to easily identify where the padding has occurred. After this path, the degrees of the tree are either three or four. Each degree-three node means that the corresponding

symbol of y is a 0, degree-four means 1. The argument that pad is a padding function is immediate. □

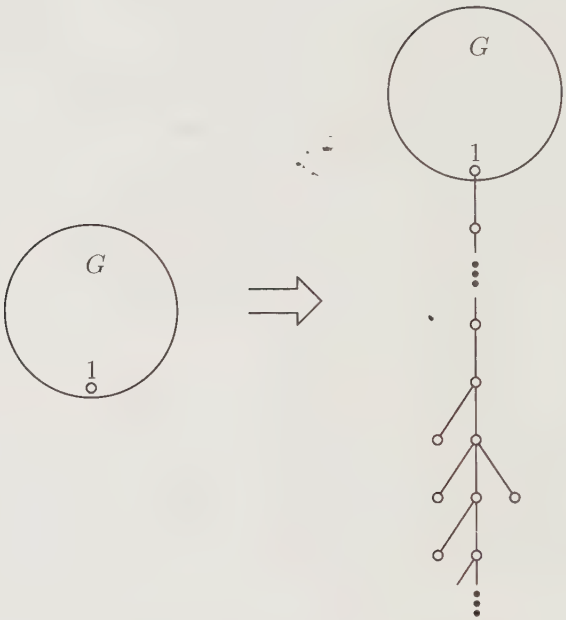


Figure 14-2. Padding for CLIQUE.

Padding functions are almost always completely trivial to construct for any of the **NP**-complete problems in this book (or anywhere for that matter; see Problems 14.5.5). And they can be used to take care of the first problem we had with reductions: That they are usually not one-to-one.

Lemma 14.1: Suppose that R is a reduction from language K to language L, and that pad is a padding function for L. Then the function mapping $x \in \Sigma^*$ to $\text{pad}(R(x), x)$ is a length-increasing one-to-one reduction. Furthermore, there is a logarithmic-space algorithm R^{-1} which, given $\text{pad}(R(x), x)$ recovers x.

Proof: That $\text{pad}(R(x), x)$ is a reduction follows easily from the fact that R is a reduction, and from properties (i) and (ii) of padding functions. That it is length-increasing follows from (iii). Finally, (iv) assures that we can recover x from $\text{pad}(R(x), x)$ in logarithmic space. □

Now that we know how to make our reductions one-to-one, length increasing, and efficiently invertible, a classical technique takes care of the rest (that is, the onto property):

Theorem 14.2: Suppose that $K, L \subseteq \Sigma^*$ are languages, and that there are reductions R from K to L and S from L to K. Suppose further that these reductions are one-to-one, length-increasing, and logarithmic-space invertible. Then K and L are polynomially isomorphic.

Proof: Since R and S are invertible, there are functions R^{-1} and S^{-1} mapping strings in L to shorter strings in K, only that R^{-1} and S^{-1} are *partial functions*, that is, they may not be defined for some strings in Σ^* (those that are not in the range of R and S, which were not assumed to be onto).

We shall now define a function h and prove that it is a polynomial-time isomorphism. Define the S-chain of a string $x \in \Sigma^*$ to be the following sequence of strings: $(x, S^{-1}(x), R^{-1}(S^{-1}(x)), S^{-1}(R^{-1}(S^{-1}(x))), \ldots)$, first applying S^{-1}, then applying alternatingly the inverse functions for as long as possible; notice that the S-chain of x could consist simply of (x), if $S^{-1}(x)$ is undefined. The R-chain of x is defined analogously. The S-chain is length-decreasing, and thus it must end somewhere, after at most $|x|$ applications. The point where the S-chain ends is either a string of the form $S^{-1}(R^{-1}(\ldots S^{-1}(x)\ldots))$ on which R^{-1} is not defined, or a string of the form $R^{-1}(S^{-1}(\ldots S^{-1}(x)\ldots))$ on which S^{-1} is undefined. *In the first case we define* $h(x) = S^{-1}(x)$, *while in the second* $h(x) = R(x)$. h is a well-defined function, because if an x falls in the first case, then its S-chain did not stop at the first step, and hence $S^{-1}(x)$ is defined; $R(x)$ is always defined.

We have to prove that h is a polynomial-time isomorphism between K and L, that is, we have to establish (i) through (iv) in Definition 14.1. To show that h is a one-to-one function, suppose that $h(x) = h(y)$ for $x \neq y$. First, since both R and S^{-1} are one-to-one, x and y must fall into different cases in the definition of h, that is $h(x) = S^{-1}(x) = R(y) = h(y)$, $y = R^{-1}(S^{-1}(x))$. Thus, the S-chain of y is a suffix of that of x, absurd since x and y were assumed to fall into different cases of the definition of h.

To prove that h is onto, consider any string $y \in \Sigma^*$. We have to show that there is an x with $h(x) = y$. Consider the R-chain of y. If it stops at an undefined R^{-1}, this means that the S-chain of $S(y)$ also stopped at an undefined R^{-1} application, and thus $h(S(y)) = S^{-1}(S(y)) = y$; we have found our $x = S(y)$. If the R-chain from y stops at an undefined S^{-1} application, then consider $x = R^{-1}(y)$. The S-chain from x stops at an undefined S^{-1} application, and so $h(x) = R(x) = R(R^{-1}(y)) = y$. We have shown (ii), that h is a bijection. Notice that the inverse of h is defined in a very symmetric way: If the R-chain from x stops at an S^{-1}, then $h^{-1}(x) = R^{-1}(x)$, otherwise $h^{-1}(x) = S(x)$.

To prove (iii), just notice that both R and S^{-1} map strings in K to strings in L, and strings not in K to strings not in L. Finally, for (iv), $h(x)$ can be computed in polynomial time by first computing the S-chain of x by up to $|x|$ computations of the polynomial-time functions R^{-1} and S^{-1} on strings shorter

than x (incidentally, this is the step that is difficult to perform in logarithmic space) and then applying once to x the appropriate function, S^{-1} or R. The argument for h^{-1} is identical. □

From Theorem 14.2, we can show that all known **NP**-complete problems are in fact polynomially isomorphic. We indicate below the kind of result that is possible:

Corollary: The following **NP**-complete languages (among many others) are polynomially isomorphic: SAT, NODE COVER, HAMILTON PATH, CLIQUE, MAX CUT, TRIPARTITE MATCHING, and KNAPSACK.

Proof: Since these problems are all **NP**-complete, there are reductions between all of them. By Lemma 14.1, it only remains to show that there is a padding function for each of these problems. For CLIQUE and SAT this is done in Examples 14.2 and 14.3. The simple constructions for the other problems are left to the reader (see Problem 14.5.5). □

Density

Polynomial isomorphism is related with an important attribute of languages, their *density*. Let $L \subseteq \Sigma^*$ be a language. Its *density* is the following function from nonnegative integers to nonnegative integers: $\text{dens}_L(n) = |\{x \in L : |x| \leq n\}|$. That is, $\text{dens}_L(n)$ is the number of strings in L of length up to n. The relationship between density and isomorphism is summarized by the following observation:

Proposition 14.1: If $K, L \subseteq \Sigma^*$ are polynomially isomorphic, then dens_K and dens_L are polynomially related.

Proof: All strings in K of length at most n are mapped by the polynomial-time isomorphism into strings of L of length at most $p(n)$, where p is the polynomial bound of the isomorphism. Since this mapping must be one-to-one, $\text{dens}_K(n) \leq \text{dens}_L(p(n))$. Similarly $\text{dens}_L(n) \leq \text{dens}_K(p'(n))$, where p' is the polynomial bound of the inverse isomorphism. □

Obviously, the density function of a language cannot grow faster than exponentially. Thus, we can distinguish two kinds of languages: The *sparse* languages, with *polynomially bounded* density functions, and the *dense* languages, with superpolynomial densities. A familiar kind of sparse languages, but by no means the only one, are the *unary languages*, subsets of $\{0\}^*$—notice that for any such language U we have $\text{dens}_U(n) \leq n$.

All **NP**-complete languages we have seen (including those sampled in the Corollary to Theorem 14.2) are dense, and so by Proposition 14.1 they cannot be polynomially isomorphic to sparse languages. This strongly suggests that *sparse languages cannot be* **NP***-complete*. There is an interesting direct argument that proves this for *unary* languages, naturally on the assumption that $\mathbf{P} \neq \mathbf{NP}$:

Theorem 14.3: Suppose that a unary language $U \subseteq \{0\}^*$ is **NP**-complete. Then $\mathbf{P} = \mathbf{NP}$.

Proof: Suppose that there is a unary language U such that there is a reduction R from SAT to U (this is what it means to be **NP**-complete). We can assume that $R(x) \in \{0\}^*$ (otherwise, whenever $R(x) \notin \{0\}^*$ we can make R output a standard string in $\{0\}^* - U$; if no such string exists, the polynomial algorithm is immediate). We shall describe a polynomial-time algorithm for SAT.

Our algorithm exploits the valuable *self-reducibility* property of SAT (recall Example 10.3 and the proof of Theorem 13.2). Given the Boolean expression ϕ with n variables $x_1, \ldots, x_n$, our algorithm considers certain partial truth assignments to the first j variables. Such a partial truth assignment is represented by a string $t \in \{0,1\}^j$, where $t_i = 1$ means that $x_i =$ **true**, and $t_i = 0$ means $x_i =$ **false**. For each such partial truth assignment t, let $\phi[t]$ be the expression resulting from ϕ if we substitute the truth values suggested by t for the first j Boolean variables in ϕ (omitting any **false** literals from a clause, and omitting any clause with a **true** literal). These expressions can be visualized as forming a "binary tree." Obviously, if $|t| = n$, then $\phi[t]$ is either **true** (it has no clauses) or **false** (it has an empty clause).

By "the self-reducibility of SAT" we essentially mean that the following recursive algorithm correctly determines whether $\phi[t]$ is satisfiable:

> If $|t| = n$ then return "yes" if $\phi[t]$ has no clauses, else return "no".
> Otherwise return "yes" if and only if either $\phi[t0]$ or $\phi[t1]$ returns "yes."

This recursive algorithm, if applied to $\phi = \phi[\epsilon]$, will successively call itself with arguments that range over all nodes of the tree. Our polynomial-time algorithm for SAT is based on the one shown above, with a clever twist—a familiar trick for speeding up recursion in general: During the recursive calls of the algorithm, we keep a "hash table" of already discovered results—that is, a list of pairs of the form $(H(t), v)$, where H is a function to be specified later, and v is the value—"yes" or "no"—of $\phi[t]$. On evaluating $\phi[t]$, we first look up $H(t)$ in the table to see if by any chance we already know the answer. The complete algorithm is this:

> If $|t| = n$ then return "yes" if $\phi[t]$ has no clauses, else return "no".
> Otherwise look up $H(t)$ in the table; if a pair $(H(t), v)$ is found return v.
> Otherwise return "yes" if either $\phi[t0]$ or $\phi[t1]$ returns "yes;"
> return "no" otherwise.
> In either case, update the table by inserting $(H(t), v)$.

The specification of H left aside, this is the full description of the algorithm.

Obviously, for this algorithm to be correct and efficient, H must be a carefully selected function with the following two properties: First, if $H(t) = H(t')$ for two partial truth assignments t and t', then $\phi[t]$ and $\phi[t']$ must be either both satisfiable or both unsatisfiable. Second, the range of H must be small, so that the table can be searched efficiently, and many invocations succeed in finding the value in the table.

But we know of such a function H: *It is the reduction R from* SAT *to the unary language U*. That is, we can define $H(t) = R(\phi[t])$. Since R is a reduction from SAT to U, the first property of H above holds: If $R(\phi[t]) = R(\phi[t'])$, then either both $\phi[t]$ and $\phi[t']$ are satisfiable (and thus $R(\phi[t]) \in U$) or both are unsatisfiable. Also, all values $H(t)$ must be of length at most $p(n)$, the polynomial bound on R, when applied to an expression with n variables. And since U is unary, there are at most $p(n)$ such values.

Let us now estimate the running time of the algorithm. Each execution of the algorithm, if we disregard for a moment the recursive calls, takes time bounded from above by $p(n)$ for the look-up of the table. Thus, the total time is $\mathcal{O}(Mp(n))$, where M is the total number of invocations of the algorithm.

Now the invocations of the algorithm form a binary tree of depth at most n. We claim that we can pick a set $T = \{t_1, t_2, \ldots\}$ of invocations, identified with their partial truth assignments, such that (a) $|T| \geq \frac{M}{2n}$; (b) all invocations in T are recursive (that is, they are not leaves of the tree), and, more important, (c) *none of the elements of T is a prefix of another element of T.*

We construct such a set T as follows: We first delete from consideration all leaves of the tree (that is, all invocations that are not recursive). Since the tree is binary, there are at least $\frac{M}{2}$ non-leaves remaining. We then select any bottom undeleted invocation t and add it to T, and we delete from further consideration all of its ancestors in the tree. Notice that the ancestors of t in the tree are all its prefixes, and so there is no way that in subsequent rounds we shall ever add to T a prefix of t. We continue like this always picking a bottom undeleted configuration, adding it to T, and deleting it and all of its ancestors, until there is no undeleted node in the tree. Since at each step we delete at most n invocations (remember, the tree has depth at most n), the resulting T has at least $\frac{M}{2n}$ independent invocations.

We now claim that all invocations in T are mapped to different H values, that is, if $t_i \neq t_j$ and $t_i, t_j \in T$, then $H(t_i) \neq H(t_j)$. The reason is simple: Since none of t_i, t_j is a prefix of another, the invocation of one started after the invocation of the other had terminated. Thus, if they had the same H value, the one that was invoked second would have looked it up, and therefore would not be recursive.

We have thus shown that there are at least $\frac{M}{2n}$ different values in the table; but we know that there are at most $p(n)$ such values. We conclude that $\frac{M}{2n} \leq p(n)$, and thus $M \leq 2np(n)$. Since the running time of the algorithm is

$\mathcal{O}(Mp(n))$, a polynomial bound of $\mathcal{O}(np^2(n))$ has been proved. □

By more sophisticated techniques, somewhat reminiscent of the proofs of Theorems 14.1 and 14.2, Theorem 14.3 can be extended to *any* sparse language (see the references in 14.5.4).

14.3 ORACLES

Analogy is our favorite reasoning method. When faced with a difficult problem, people have the tendency to ponder how the same problem would be tackled in another situation, in an "alternative universe."

It is possible to do this with complexity questions, like $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$. But what is an "alternative universe" in the context of complexity? Here is a simple proposal: Our world can be characterized by the cruel fact that no computation whatsoever is free. But we could conceive worlds where certain computations do come for the asking. For example, we can imagine an algorithm which, once in a while during its computation, asks whether a Boolean expression it has constructed is satisfiable or not, *and gets an instantaneous correct answer.* This answer is then used for the continuation of the computation, perhaps in the construction of the next query expression; and so on. This is *the world of* SAT, where a friendly "oracle" answers all our SAT queries for free. Naturally, it is a rather unrealistic and far-fetched world but remember, this was a search for "alternative universes."

Once we have defined such a world, we can ask whether $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ in it. In the world of SAT it so turns out that this is not so easy to tell. Although polynomial algorithms in this world are very powerful, and can solve any problem in *our* **NP** effortlessly, the oracle also boosts the power of nondeterministic machines to new mysterious heights, explored in Chapter 17. The $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ question in the world of SAT is perhaps even harder to resolve than the one in our world. However, in this section we shall construct alternative universes where the $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ question is easy. In fact, we shall construct two universes *in which this question has two opposing answers.*

But we must now define algorithms with oracles:

Definition 14.3: A *Turing machine $M^?$ with oracle* is a multi-string deterministic Turing machine that has a special string called the *query string*, and three special states $q_?$, the *query state*, and $q_{\text{YES}}, q_{\text{NO}}$, the *answer states*. Notice that we define $M^?$ independently of the oracle used; the "?" in the exponent indicates that any language can be "plugged in" as an oracle.

Let $A \subseteq \Sigma^*$ be an arbitrary language. The computation of oracle machine $M^?$ with oracle A proceeds like in an ordinary Turing machine, except for transitions from the query state. From the query state $M^?$ moves to either q_{YES} or q_{NO} *depending on whether the current query string is in A or not.* The answer states allow the machine to use this answer in its further computation.

The computation of $M^?$ with oracle A on input x is denoted $M^A(x)$.

Time complexity for Turing machines with oracles is defined precisely the same way as with ordinary Turing machines. In fact, this is exactly why such machines are so unrealistic: Each query step counts as one ordinary step. Also, nondeterministic Turing machines with oracles can be defined in the same way. (For a difficulty associated with defining space complexity for oracle machines see 14.5.8). Thus if $\mathcal{C}$ is any deterministic or nondeterministic time complexity class, we can define $\mathcal{C}^A$ to be the class of all languages decided (or accepted) by machines of the same sort and time bound as in $\mathcal{C}$, only that the machines have now oracle A. □

We shall now show our first result:

Theorem 14.4: There is an oracle A for which $\mathbf{P}^A = \mathbf{NP}^A$.

Proof: We have already argued that SAT is not quite the A we are looking for. But what exactly *are* we looking for? We are seeking a language whose help would render nondeterminism powerless. Or rather, a language that will make polynomial computation so powerful that nondeterminism has nothing to add. And we know a place where nondeterminism is no more powerful than determinism: *Polynomial space-bounded computation.* By Savitch's theorem (recall Section 7.3), nondeterministic and deterministic polynomial space coincide.

Take A to be any **PSPACE**-complete language†. We have

$$\mathbf{PSPACE} \subseteq \mathbf{P}^A \subseteq \mathbf{NP}^A \subseteq \mathbf{NPSPACE} \subseteq \mathbf{PSPACE}.$$

Hence, $\mathbf{P}^A = \mathbf{NP}^A$. The first inclusion holds because A is **PSPACE**-complete, and thus any language $L \in \mathbf{PSPACE}$ can be decided by a polynomial-time deterministic Turing machine that performs the reduction from L to A, and then uses the oracle once. The second inclusion is trivial. For the third, any nondeterministic polynomial-time Turing machine with oracle A can be simulated by a nondeterministic polynomial space-bounded Turing machine, which resolves the queries to A by itself, in polynomial space. The last inclusion is Savitch's theorem. □

An oracle for achieving the opposite goal is a little more subtle.

Theorem 14.5: There is an oracle B for which $\mathbf{P}^B \neq \mathbf{NP}^B$.

Proof: What is needed now is an oracle B that enhances the power of nondeterminism, plus a language that takes maximum advantage of B and of nondeterminism, so that $L \in \mathbf{NP}^B - \mathbf{P}^B$. We define the language L first:

$$L = \{0^n : \text{There is a } x \in B \text{ with } |x| = n\},$$

† Strictly speaking, we have yet not shown that PSPACE-complete languages exist. But they do exist; the reader should at this point be able to define such a language and prove it complete; see also problem 8.4.4. In any event, in Chapter 19 we show many PSPACE-completeness results—not using Theorem 14.4, honest...

a unary language. Notice immediately that $L \in \mathbf{NP}^B$: A nondeterministic machine with oracle B could guess on input 0^n an x with length n, and use its oracle to verify that $x \in B$.

We shall now have to define $B \subseteq \{0,1\}^*$ in such a way that $L \notin \mathbf{P}^B$. This is done by some form of "slow diagonalization" over all deterministic polynomial-time Turing machines with oracle. Suppose that we have an enumeration $M_1^?, M_2^?, \ldots$ of all such machines. We assume that every machine appears in the enumeration *infinitely many times*. Any reasonable enumeration satisfies this useful condition, since any machine can be "padded" with useless states in arbitrarily many ways that do not affect the language decided, and all these variants must appear somewhere in the enumeration.

We define B in stages. At the beginning of the ith stage we have already computed B_{i-1}, the set of all strings in B of length less than i. We also have a set $X \subseteq \Sigma^*$ of *exceptions*, strings that we must remember not to put in B (initially, at the beginning of the first stage, $B_0 = X = \emptyset$). We are about to define B_i, that is, to decide which strings of length i should be in B. We do this by simulating $M_i^B(0^i)$ for $i^{\log i}$ steps. Notice the number of steps: It is chosen to be a function much smaller than exponential, and still asymptotically larger than any polynomial.

During the simulation of $M_i^B(0^i)$, the oracle machine may ask a query of the form "is $x \in B$?". How do we answer the query? If $|x| < i$, then we simply look up x in B_{i-1}. If it is in B_{i-1} we answer "yes," that is, M_i^B goes to state q_{YES}, otherwise to q_{NO}, and the computation continues. But if $|x| \geq i$, then M_{i-1} goes to state q_{NO}—we answer "no"—and we add x to the set X of exceptions—to remember our commitment that $x \notin B$.

Suppose that in the end, after $i^{\log i}$ or fewer steps, the machine rejects. Remember that we want to prevent M_i^B from deciding L. To this end, we define B_i to be $B_{i-1} \cup \{x \in \{0,1\}^* : |x| = i, x \notin X\}$. This way we make sure that $0^i \in L$ (since there is a string of length i in B, recall the definition of L), and M_i^B, which has rejected 0^i, fails to be the machine that decides L: $L(M_i^B) \neq L$. But how do we know that the set $\{x \in \{0,1\}^* : |x| = i, x \notin X\}$ is non-empty? We know this because X contains no more than $\sum_{j=1}^{i} j^{\log j}$ strings of length i (this is the total number of steps simulated so far on all machines), and a little calculation shows that this sum is always smaller than 2^i, the number of strings in $\{0,1\}^i$. If M_i^B accepts 0^i in the allotted time, then we set $B_i = B_{i-1}$. Again, this makes sure that $0^i \notin L$, and therefore $L(M_i^B) \neq L$.

But what if $M_i^B(0^i)$ fails to halt after $i^{\log i}$ steps? After all, the polynomial bound $p(n)$ of the machine may be so large, and this particular value of i so small, that $i^{\log i} < p(i)$. If this happens, we let $B_i = B_{i-1}$, as if the machine had accepted. But of course this way we have made no progress towards ensuring that $L(M_i^B) \neq L$. The point is that *machines equivalent to this one will appear infinitely often in our enumeration*, and so one must eventually appear as M_I^B,

where the index I is so large that $I^{\log I} \geq p(I)$. This will ensure that $L \neq L(M_I^B) = L(M_i^B)$.

Continuing this way, we completely define the oracle B. Since this definition of B systematically rules out all polynomial-time machines with oracle B from deciding L, we must conclude that $L \notin \mathbf{P}^B$, and the proof is complete. □

This interesting pair of results (Theorems 14.4 and 14.5) has several important methodological implications. First, our original idea of reasoning by analogy leads nowhere: Analogy can give us all kinds of contradictory answers. Second, these results are a warning: The $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ question will not be settled by a proof that can be carried over to oracle machines, that is, a proof technique that *transcends worlds*. And many of the techniques we have used so far in this book *can* be carried over verbatim from one world to another (see the next example).

Oracle results are very useful tools of "exploratory research" in complexity. Suppose that we are wondering about a complexity question, such as whether $\mathcal{C} \stackrel{?}{=} \mathcal{D}$, for two complexity classes $\mathcal{C}$ and $\mathcal{D}$ —the reader has seen plenty of examples already, with many more to come. An oracle B for which $\mathcal{C}^B \neq \mathcal{D}^B$ is an important indication that $\mathcal{C} \neq \mathcal{D}$ *is a legitimate possibility*, that there is probably no trivial proof of $\mathcal{C} = \mathcal{D}$ waiting to be observed. Naturally, we may try to also show that $\mathcal{C}^A = \mathcal{D}^A$ for some oracle A; but this is almost always immediate: The **PSPACE**-complete oracle A of Theorem 14.4 collapses not only **NP** with **P**, but all classes between **P** and **PSPACE**, so it is likely to also identify $\mathcal{C}$ with $\mathcal{D}$.

Thus oracle results help establish complexity questions such as $\mathcal{C} \stackrel{?}{=} \mathcal{D}$ as meaningful, nontrivial conjectures. We can even apply this technique to more involved situations, where class collapse is not the only issue. Examples: "is it possible that $\mathbf{NP} = \mathbf{coNP}$, and still $\mathbf{P} \neq (\mathbf{NP} \cap \mathbf{coNP})$?" or even "does **BPP** have complete problems?" (see the references of this and the next chapters for a host of such oracle results).

Example 14.4: We have already mentioned that the importance of oracle results lies with the fact that many "ordinary" proof techniques in complexity are not affected by the addition of oracles in the computation, and so an oracle result is a warning that such ordinary techniques will not suffice to prove the opposite statement. This has to be taken with a grain of salt, and is hard to quantify, so let us examine some simple cases.

In the proof of Theorem 14.4 the reader probably did not even notice the claim $\mathbf{P}^A \subseteq \mathbf{NP}^A$. It is a true statement, because the simple argument used to show $\mathbf{P} \subseteq \mathbf{NP}$ (a deterministic Turing machine is a special case of a nondeterministic one) still holds for oracle machines. Such simple arguments can be carried over across universe boundaries easily.

But let us now examine if the argument in Theorem 14.1 is also easily

transferable. Restated relative to an oracle A, the theorem should read something like "if $\mathbf{P}^A \neq \mathbf{NP}^A$, then there is a language in $\mathbf{NP}^A - \mathbf{P}^A$ which is not $\mathbf{NP}^A$-complete." But when we say "$\mathbf{NP}^A$-complete," do we also allow the use of oracles in our reductions? With a little reflection we may decide that the right way to state the result is in terms of ordinary reductions (especially since there are difficulties associated with defining space-bounded oracle machines, see the references).

Do all steps of the proof then translate easily to the oracle case? The proof is based on the design of a language L in terms of an algorithm F. All simulations seem to translate verbatim in the world of any oracle A, except for one worrisome point: The simulation, as well as the definition of L, use a machine S that decides SAT. And SAT is particular to *our* world, it is complete for *our* **NP**, and not for $\mathbf{NP}^A$. There seems to be no reasonable definition of SAT^A.

To get around this difficulty, we must replace the usage of SAT with an $\mathbf{NP}^A$-complete problem. Such problems exist; for example:

$$C^A = \{(M^A, x) : \text{nondeterministic oracle machine } M^A \text{ accepts } x \text{ in time } |x|\}$$

never fails to be one. With this modification, the proof works as intended.

Translating Theorem 14.3 on unary **NP**-complete sets into the oracle context is even harder, because that proof relies so much on SAT and its self-reducibility.

A most important complexity result that makes heavy use of complete problems is Proposition 13.4, the weak verifier version of Cook's theorem. As it turns out, there are oracles under which *this important result does not hold.*

There is a point to be made here: An important class of proof techniques that do not translate immediately to oracle machines (and are thus good candidates for resolving complexity questions for which we have adverse oracle results) are techniques based on *complete problems.* We shall see more uses of such techniques in future chapters—including one remarkable proof that identifies two complexity classes, **PSPACE** and the **IP** class defined in Section 12.2; for these two classes a separating oracle had been known. □

14.4 MONOTONE CIRCUITS

In Chapter 11 we briefly discussed circuit complexity, and formulated Conjecture B, a strengthening of the $\mathbf{P} \neq \mathbf{NP}$ conjecture, stating that **NP**-complete problems have no polynomial circuits (uniform or not).

Progress in proving Conjecture B—indeed, proving lower bounds for the circuit complexity of *any* problem, in **NP** or not—has been *very* slow. Despite Theorem 4.3 stating that there are Boolean functions requiring as much as $\frac{2^n}{2n}$ gates to be computed (in fact, almost all Boolean functions do), at present the

largest lower bounds we have been able to prove for explicit families of functions are of the form $k \cdot n$ for small constants k (see the references in 11.5.26).

In view of the apparent difficulty in proving Conjecture B, we may try to prove something weaker. We could try to prove exponential lower bounds for the circuit complexity of **NP**-complete problems in a *weaker* circuit model. And we have seen a most natural weaker circuit model: The *monotone circuits*, that is, ones without NOT gates. Monotone circuits are expressive enough to have a **P**-complete CIRCUIT VALUE problem (recall Corollary 2 to Theorem 8.1). Naturally, as we have already observed, monotone circuits can only compute *monotone functions* (Boolean functions whose output cannot change from **true** to **false** when one input changes the other way). Many **NP**-complete problems, such as BISECTION WIDTH, NODE COVER, and KNAPSACK are not monotone, and so they cannot be computed by monotone circuits, however large. But other important **NP**-complete problems, such as HAMILTON PATH and CLIQUE, are indeed monotone (turning any bit of the adjacency matrix from **false** to **true** cannot change the answer from **true** to **false**), and thus they certainly have monotone circuits that compute them (recall Problem 4.4.13). The question is, *how small can these monotone circuits be?*

Let us take CLIQUE, surely an **NP**-complete problem (Corollary 2 to Theorem 9.4). By $\text{CLIQUE}_{n,k}$ we shall understand the Boolean function deciding whether a graph $G = (V, E)$ with n nodes has a clique of size k. The input gates correspond to the entries of the adjacency matrix of G; that is, there are $\binom{n}{2}$ input gates, and input gate $g_{[i,j]}$ is set to **true** if and only if $[i, j] \in E$. $\text{CLIQUE}_{n,k}$ is a monotone function, and thus it can be computed by a monotone circuit. Here is one such circuit: For each set $S \subseteq V$ with $|S| = k$ we have a subcircuit, with $\mathcal{O}(k^2)$ AND gates, testing whether S forms a clique. We repeat this for all $\binom{n}{k}$ subsets $S_1, S_2 \ldots, S_{\binom{n}{k}}$ of k nodes, and take a big OR of the outcomes. This is a monotone circuit computing the $\text{CLIQUE}_{n,k}$ function, with $\mathcal{O}(k^2\binom{n}{k})$ gates.

We call a circuit such as the one described above, that tests whether a family of subsets of V form a clique, and returns **true** if any one of the sets in the family does, a *crude circuit*. For example, the above crude circuit will be denoted $\text{CC}(S_1, \ldots, S_{\binom{n}{k}})$, which means that it computes the OR of $\binom{n}{k}$ subcircuits, each indicating whether the corresponding set in the list is a clique. In general, the sets tested by a crude circuit $\text{CC}(X_1, \ldots, X_m)$ can be arbitrary subsets of V, not necessarily of cardinality k.

Although the crude monotone circuit described above has polynomial size when k is a constant, it is exponentially large when k becomes, say, $\sqrt[4]{n}$. The following result states that this exponential dependence is inherent.

Theorem 14.6 (Razborov's Theorem): There is a constant c such that for large enough n all monotone circuits for $\text{CLIQUE}_{n,k}$ with $k = \sqrt[4]{n}$ have size at

least $2^{c\sqrt[8]{n}}$.

The proof of this remarkable result proceeds along the following path: We shall describe a way of *approximating* any monotone circuit for $\text{CLIQUE}_{n,k}$ by a restricted kind of crude circuit. The approximation will proceed in steps, one step for each gate of the monotone circuit. We shall show that, although each step introduces rather few errors (false positives and false negatives, this is shown in Lemmata 14.3 and 14.4), *the crude circuit that results from this process has exponentially many errors* (this is shown in Lemma 14.5). We must conclude that the approximation takes exponentially many steps, and thus the original monotone circuit for $\text{CLIQUE}_{n,k}$ has exponentially many gates.

Recall that $k = \sqrt[4]{n}$. Define ℓ to be $\sqrt[8]{n}$. p and M are integers to be fixed later in the proof; for the time being, suffice it to say that p is also about $\sqrt[8]{n}$, while $M = (p-1)^\ell \ell!$, exponentially large in n. Also, by the values of k and ℓ, it is easy to see that $2\binom{\ell}{2} \leq k$. Each crude circuit used in the approximation process is of the form $\text{CC}(X_1, \ldots, X_m)$, where the X_i's are subsets of V with at most ℓ nodes each, and there are at most M X_i's ($m \leq M$).

We must show how to approximate any circuit for $\text{CLIQUE}_{n,k}$ by such a crude circuit. We shall do this inductively: Since any monotone circuit can be considered the OR or AND of two subcircuits, we shall show how to build approximators of the overall circuit from the approximators of the two subcircuits (the induction is easy to start, because each input gate g_{ij} denoting whether $[i,j] \in E$ can indeed be seen as a crude circuit $\text{CC}(\{i,j\})$). That is, given two crude circuits $\text{CC}(\mathcal{X})$ and $\text{CC}(\mathcal{Y})$, where $\mathcal{X}$ and $\mathcal{Y}$ are families of at most M sets of nodes, each set with at most ℓ nodes, we shall show how to construct the approximate OR and the approximate AND of these circuits.

We start with the OR. Basically, the approximate OR of $\text{CC}(\mathcal{X})$ and $\text{CC}(\mathcal{Y})$ is $\text{CC}(\mathcal{X} \cup \mathcal{Y})$; that is, we take the union of the two families. So far, this is no approximation at all, the new circuit is equivalent to the OR of the other two. But of course there is a problem: There may now be more than M sets in the family, and we must find a way to reduce the number of sets to M or less. *At the heart of the proof is a sophisticated, systematic way for reducing the size of families of sets*, called *plucking*. We explain it next.

A *sunflower* is a family of p sets $\{P_1, \ldots, P_p\}$, called *petals*, each of cardinality at most ℓ, such that all pairs of sets in the family have the same intersection (called *the core* of the sunflower). The following lemma shows that any large enough family of sets has a sunflower:

Lemma 14.2 (The Erdős-Rado Lemma): Let $\mathcal{Z}$ be a family of more than $M = (p-1)^\ell \ell!$ nonempty sets, each of cardinality ℓ or less. Then $\mathcal{Z}$ must contain a sunflower.

Proof: Induction on ℓ. For $\ell = 1$ the statement is that p different singletons form a sunflower, which they do.

So, suppose that $\ell > 1$. Consider a maximal subset of $\mathcal{Z}$, call it $\mathcal{D}$, of disjoint sets (that is, every set in $\mathcal{Z} - \mathcal{D}$ intersects some set in $\mathcal{D}$). If $\mathcal{D}$ contains at least p sets, then it constitutes a sunflower with empty core, and we are done. Otherwise, let D be the union of all sets in $\mathcal{D}$. Since $\mathcal{D}$ contains fewer than p sets, we know that $|D| \leq (p-1)\ell$. Furthermore, we know that D intersects every set in $\mathcal{Z}$. Since $\mathcal{Z}$ has more than M sets, and each intersects some element of D, there is an element of D that intersects more than $\frac{M}{(p-1)\ell} = (p-1)^{\ell-1}(\ell-1)!$ sets in $\mathcal{Z}$; call this element d. Consider now a new family of sets,

$$\mathcal{Z}' = \{Z - \{d\} : Z \in \mathcal{Z} \text{ and } d \in Z\}.$$

We know that $\mathcal{Z}'$ has more than $M' = (p-1)^{\ell-1}(\ell-1)!$ sets, and thus by induction (notice that M' is just M with ℓ decreased by one) it contains a sunflower, say $\{P_1, \ldots, P_p\}$. Then the following is a sunflower in $\mathcal{Z}$, completing the proof: $\{P_1 \cup \{d\}, \ldots, P_p \cup \{d\}\}$. □

By this lemma, whenever we have a family of more than M sets, we can always find a sunflower in it. Now, *plucking a sunflower* entails replacing the sets in the sunflower by its core. Thus, whenever we have more than M sets in a family, we can reduce their number to M or less by repeatedly finding a sunflower and plucking it. If finally this cannot be done any more, by the lemma above we know that we have no more than M sets. If $\mathcal{Z}$ is a family of sets, we denote the result of this repeated plucking applied to $\mathcal{Z}$ pluck($\mathcal{Z}$).

To return to our proof, the approximate OR of two crude circuits CC($\mathcal{X}$) and CC($\mathcal{Y}$) is defined to be CC(pluck($\mathcal{X} \cup \mathcal{Y}$)).

The approximate AND of two crude circuits CC($\mathcal{X}$) and CC($\mathcal{Y}$) is defined as follows:

$$\text{CC}(\text{pluck}(\{X_i \cup Y_j : X_i \in \mathcal{X}, Y_j \in \mathcal{Y}, \text{ and } |X_i \cup Y_j| \leq \ell\}).$$

That is, in order to construct the approximate AND of two crude circuits we take all possible cross unions, we delete all sets that have more than ℓ elements, and pluck the remaining family as far as possible.

We shall next argue that these stepwise approximations are reasonable approximations, in that *they introduce few errors*. In our analysis we shall only monitor the bahavior of the approximator circuits on some very specialized input graphs, called the *positive examples* and the *negative examples*. A positive example is simply a graph that has $\binom{k}{2}$ edges connecting k nodes in all possible ways, and no other edges. Obviously, there are $\binom{n}{k}$ such graphs, and they all should elicit a **true** output from CLIQUE$_{n,k}$.

The negative examples are outcomes of the following experiment: Color the nodes with $k-1$ different colors. Then join by an edge any two nodes that are colored differently. These are all the edges of the graph. It is not hard to

see that such a graph has no k-clique (because it is $(k-1)$-colorable). There are $(k-1)^n$ negative examples overall. In what follows, we shall be counting the positive and negative examples for which our approximators are in error. Although two colorings may produce the same graph (for example, if the names of two colors are interchanged), in our accounting we shall consider two distinct colorings to be two distinct negative examples.

Consider two crude circuits and the approximator of their OR, as defined above. Suppose that, when a positive example E is supplied as input to the two original crude circuits, at least one of them returns **true**; and still, the approximator of their OR returns **false** on E. We say that *the approximation has introduced a false negative*. Similarly, if a negative example returns **false** on both crude circuits, but **true** on the approximator of their OR, we say that *the approximation has introduced a false positive*. Also, the AND approximator introduces a false negative if for some positive example both constituent crude circuits compute **true**, but the resulting crude circuit computes **false**. A false negative is introduced if for some coloring at least one of the constituent crude circuits returns **false** but the approximator of their AND returns **true**. The question is, *how many false positives and false negatives are introduced by each approximation step?*

Lemma 14.3: Each approximation step introduces at most $M^2 2^{-p}(k-1)^n$ false positives.

Proof: Consider first an approximation step for an OR, and in particular one of the possibly many pluckings involved, say the replacement of sunflower $\{Z_1, \ldots, Z_p\}$ by its core Z. What is a false positive that is introduced by this plucking? It is a coloring such that there is a pair of identically colored nodes in each petal (and so both crude circuits returned **false**), but at least one node from each pair was plucked away, and the core is all different colors. How many such colorings are there?

This question is easier to answer if rephrased this way: What is the probability that, if a coloring of the vertices in V is chosen at random, all Z_i's have repeated colors, but Z does not? Let $R(X)$ stand for the event that there are repeated colors in set X. We have:

$$\begin{aligned}\mathbf{prob}[R(Z_1) \wedge \ldots \wedge R(Z_p) \wedge \neg R(Z)] &\leq \mathbf{prob}[R(Z_1) \wedge \ldots \wedge R(Z_p) | \neg R(Z)] \\ &= \prod_{i=1}^{p} \mathbf{prob}[R(Z_i) | \neg R(Z)] \\ &\leq \prod_{i=1}^{p} \mathbf{prob}[R(Z_i)]\end{aligned}$$

The first inequality holds because the left-hand side is actually equal to the right-hand side divided by $\mathbf{prob}[\neg R(Z)] < 1$ (this is the definition of conditional

probability). The second equality is true because the only common vertices the Z_i's have are in Z, and, given that there are no repeated colors in Z, the probabilities of repeated colors in the Z_i's are independent. The last inequality holds because the probability of repetitions in Z_i is obviously decreased if we restrict ourselves to colorings with no repetitions in $Z \subseteq Z_i$.

Consider two nodes in Z_i. The probability that they have the same color is obviously $\frac{1}{k-1}$. Since $R(Z_i)$ means that at least one of the $\binom{|Z_i|}{2}$ pairs of nodes in Z_i have the same color, it follows that $\mathbf{prob}[R(Z_i)]$ is at most $\frac{\binom{|Z_i|}{2}}{k-1} \leq \frac{\binom{\ell}{2}}{k-1} \leq \frac{1}{2}$, and thus the probability that a randomly chosen coloring is a new false positive is at most 2^{-p}. Since there are $(k-1)^n$ different colorings, we conclude that each plucking introduces $2^{-p}(k-1)^n$ false positives. Finally, since the approximation step entails up to $\frac{2M}{p-1}$ pluckings (each plucking decreases the number of sets by $p-1$, and there are no more than $2M$ sets when we start), the lemma holds for the OR approximation step.

Consider now an AND approximation step of crude circuits $\mathrm{CC}(\mathcal{X})$ and $\mathrm{CC}(\mathcal{Y})$. It can be broken down in three phases: First, we form $\mathrm{CC}(\{X \cup Y : X \in \mathcal{X}, Y \in \mathcal{Y}\})$; this introduces no false positives, because any graph in which $X \cup Y$ is a clique must have a clique in both X and Y, and thus it was accepted by both constituent crude circuits. The second phase omits from the approximator circuit several sets (those of cardinality larger than ℓ), and can therefore introduce no false positives. The third phase entails a sequence of fewer than M^2 pluckings, during each of which, by the analysis of the OR case above, at most $2^{-p}(k-1)^n$ false positives are introduced. The proof of the lemma is complete. □

Lemma 14.4: Each approximation step introduces at most $M^2\binom{n-\ell-1}{k-l-1}$ false negatives.

Proof: Plucking can introduce no false negatives, since replacing a set in a crude circuit by a subset can only increase the accepted graphs (it makes the test less stringent). Since the approximation of an OR entails only plucking, it introduces no false negatives.

Let us then consider the approximation of an AND. In the first phase we replace the conjunction of $\mathrm{CC}(\mathcal{X})$ and $\mathrm{CC}(\mathcal{Y})$ by $\mathrm{CC}(\{X \cup Y : X \in \mathcal{X}, Y \in \mathcal{Y}\})$. If a positive example is accepted by both $\mathrm{CC}(\mathcal{X})$ and $\mathrm{CC}(\mathcal{Y})$, it must be the case that its clique contains one set in $\mathcal{X}$ and one set in $\mathcal{Y}$; but then it contains the union of these sets, and thus it is accepted by the new circuit. Hence there are no false negatives yet. We next delete all sets that are larger than ℓ. Each such deletion of a set Z may introduce several false negatives, namely the cliques that contain Z. How many such cliques are there? The answer is $\binom{n-|Z|}{k-|Z|}$, and, since we know that $|Z| > \ell$, at most $\binom{n-\ell-1}{k-\ell-1}$ false negatives are introduced by each deletion. Since there are at most M^2 sets to be deleted, the lemma has

been proved. □

Lemmata 14.3 and 14.4 show that each approximation step introduces "few" false positives and false negatives. We next show that the resulting crude circuit must have "a lot" of one of one or the other:

Lemma 14.5: Every crude circuit either is identically **false** (and thus is wrong on all positive examples), or outputs **true** on at least half of the negative examples.

Proof: If the crude circuit is not identically **false**, then it accepts at least those graphs that have a clique on some set X of nodes, with $|X| \leq \ell$. But we know from the proof of Lemma 14.3 that at least half of the colorings assign different colors to the nodes in X, and thus half of the negative examples have a clique at X and are accepted. □

The proof of Razborov's theorem is now almost complete: Let us define $p = \sqrt[8]{n} \log n$, $\ell = \sqrt[8]{n}$, and thus $M = (p-1)^\ell \ell! < n^{\frac{1}{3}\sqrt[8]{n}}$ for large enough n. Since each approximation step introduces at most $M^2\binom{n-\ell-1}{k-l-1}$ false negatives, if the final crude circuit is identically **false** we must conclude that all positive examples were introduced as false negatives at some step, and thus the original monotine circuit for CLIQUE$_{n,k}$ had at least

$$\frac{\binom{n}{k}}{M^2\binom{n-\ell-1}{k-\ell-1}}$$

gates. This is at least $\frac{1}{M^2}(\frac{n-\ell}{k})^\ell$, which is at least $n^{c\sqrt[8]{k}}$, with $c = \frac{1}{12}$. Otherwise, Lemma 14.5 states that there are at least $\frac{1}{2}(k-1)^n$ false positives, and, since each approximation step introduces at most $M^2 2^{-p}(k-1)^n$ of them, we again conclude that the original monotone circuit had at least $2^{p-1}M^{-2} > n^{c\sqrt[8]{k}}$ gates, with $c = \frac{1}{3}$. □

Razborov's theorem inspires some serious hope: All we have to do now in order to prove that $\mathbf{P} \neq \mathbf{NP}$ is to establish the following:

Conjecture C: All monotone languages in **P** have polynomial monotone circuits.

Unfortunately, Conjecture C is false: Similar techniques as those used in the previous proof establish that MATCHING, for example, has no polynomial monotone circuits (see the references in 14.5.11).

14.5 NOTES, REFERENCES, AND PROBLEMS

14.5.1 Theorem 14.1 was shown in

- R. E. Ladner "On the structure of polynomial time reducibility," *J.ACM, 22*, pp. 155–171, 1975,

among several other results establishing that, under the assumption that $\mathbf{P} \neq \mathbf{NP}$, reductions create a very dense and complex structure of incomparable classes of equivalent problems within (and around) **NP**. It is worth noting that no "natural" problem has ever been shown to belong to such an intermediate class if $\mathbf{P} \neq \mathbf{NP}$. (Incidentally, although it is tempting to consider this possibility every time a particular problem in **NP** resists our first attempts at developing a polynomial algorithm and at proving it **NP**-complete, we strongly advise against it.) For a generalization of Ladner's theorem, so that it is applicable to other complexity classes, see

- U. Schöning "A uniform approach to obtain diagonal sets in complexity classes," *Theor. Computer Science 18*, pp. 95–103, 1982.

14.5.2 For classes such as **P** and **NP** it is trivial to give a recursive enumeration of all languages in the class, represented by the corresponding machines. For classes such as $\mathbf{NP} \cap \mathbf{coNP}$ and **BPP** this is not at all obvious.

Problem: Give a recursive enumeration of *all* **NP**-*complete languages*, that is, all nondeterministic polynomial Turing machines which happen to decide **NP**-complete languages. (This is from

- L. Landweber, R. J. Lipton, and E. Robertson "On the structure of the sets in NP and other complexity classes," *Theor. Comp. Science 15*, pp. 181–200, 1981.)

14.5.3 Here is a disturbing possibility: Suppose that $\mathbf{P} = \mathbf{NP}$ is proved via a *highly nonconstructive proof*; that is, although it follows from the proof that a polynomial algorithm for SAT exists, we have no clue how to explicitly state it and run it (recall the non-constructive proof of Theorem 11.6 for an idea about the possibilities here).

Problem: (a) *Give explicitly* an algorithm for SAT which has the following property: There is a polynomial $p(n)$ such that, in the event that (1) the input x is a satisfiable expression; *and* (2) $\mathbf{P} = \mathbf{NP}$, the algorithm terminates within time $p(|x|)$ with a satisfying truth assignment for x. The algorithm may behave in any way whatsoever, including divergence, if any of the two conditions is not met.

(b) Show that without the proviso (2) such an algorithm cannot exist, unless of course $\mathbf{P} = \mathbf{NP}$.

14.5.4 The issue of isomorphism between **NP**-complete problems was raised in

- L. Berman and J. Hartmanis "Isomorphism and density of NP and other complete sets," *SIAM J. Computing, 6*, pp. 305–322, 1977,

where it was observed that all known **NP**-complete languages are isomorphic (and Theorem 14.2 was proved). More importantly, it was conjectured in that paper that *all* **NP**-*complete languages* (under polynomial-time, not logarithmic space, reductions) are isomorphic. This would imply that $\mathbf{P} \neq \mathbf{NP}$, because otherwise all languages in **P** would be **NP**-complete and hence all isomorphic—and this includes both infinite

and finite languages, which is absurd. By Proposition 14.1, of course, this *isomorphism conjecture* would also imply that no sparse language can be **NP**-complete (by polynomial-time reductions), unless $\mathbf{P} = \mathbf{NP}$. But this latter implication has been proved directly, without assuming the isomorphism conjecture:

- S. R. Mahaney "Sparse complete sets for NP: Solution of a conjecture by Berman and Hartmanis," *J.CSS, 25*, pp. 130–143, 1982.

Theorem 14.3 about unary languages was an important precursor to this result, and is from

- P. Berman "Relationship between the density and deterministic complexity of NP-complete languages," *Proc. 5th Intern. Colloqu. on Automata, Languages, and Programming*, pp. 63–71, Lecture Notes in Computer Science 62, Springer Verlag, 1978.

14.5.5 Problem: Give padding functions for the following problems: KNAPSACK, MAX CUT, and EUCLIDEAN TSP (recall Problem 9.5.15).

14.5.6 Theorems 14.4 and 14.5 were proved in

- T. Baker, J. Gill, and R. Solovay "Relativizations of the P$\stackrel{?}{=}$NP question," *SIAM J. Computing 4*, pp. 431–442, 1975.

It was also shown that there are suitable oracles for all conceivable eventualities concerning **P** and **NP**; for example, there is an oracle C for which $\mathbf{NP}^C = \mathbf{coNP}^C$ but $\mathbf{P}^C \neq \mathbf{NP}^C$. Also, there are oracles D, E for which $\mathbf{NP}^D \neq \mathbf{coNP}^D$ and $\mathbf{NP}^E \neq \mathbf{coNP}^E$, but for which $\mathbf{P}^D = \mathbf{NP}^D \cap \mathbf{coNP}^D$ and $\mathbf{P}^E \neq \mathbf{NP}^E \cap \mathbf{coNP}^E$.

Another important question, whether $\mathbf{NP} \cap \mathbf{coNP}$ has complete problems, also relativizes both ways: There are oracles under which it has complete problems (this is trivial, any oracle for which $\mathbf{P} = \mathbf{NP}$ would suffice), and others under which it does not, see

- M. Sipser "On relativization and the existence of complete sets," *Proc. 9th Int. Colloqu. on Automata, Languages, and Programming*, pp. 523–531, Lecture Notes in Computer Science Vol. 140, Springer Verlag, 1982.

The same is true of other "semantic" classes like **RP** and **BPP** (recall Section 11.2), and the class **UP** (Section 12.1). For the latter result see

- J. Hartmanis and L. Hemachandra "Complexity classes without machines: On complete languages for UP," *Theor. Computer Sci.*, pp. 129–142, 1988.

14.5.7 In view of Theorems 14.4 and 14.5 it may appear that the possibilities $\mathbf{P} = \mathbf{NP}$ and $\mathbf{P} \neq \mathbf{NP}$ are tied in this respect: Both are supported by at least one oracle. However, the following result can be shown: Of all possible oracles, only an insignificant fraction (of measure zero) supports $\mathbf{P} = \mathbf{NP}$:

- C. Bennett and J. Gill "Relative to a random oracle P$\neq$NP$\neq$coNP with probability 1," *SIAM J. Comp., 10*, pp. 96–103, 1981.

This may boost our confidence that $\mathbf{P} \neq \mathbf{NP}$ is the right answer. In fact, the "random oracle hypothesis" was proposed in this paper as an intriguing generalization of the

P $\neq$ **NP** conjecture: *Two complexity classes differ if and only if almost all oracles make them differ.* This conjecture was essentially disproved in

- S. A. Kurtz "On the random oracle hypothesis," *Information and Control, 57*, pp. 40–47, 1983.

Bennet and Gill also showed in the same paper above that for most oracles **P** = **ZPP** = **RP** $\neq$ **NP** (recall Section 11.2). It has been shown elsewhere that essentially all possible combinations of containments of the classes **RP**, **ZPP**, **UP**, **NP** and their complements, as long as they are compatible with our meager knowledge of the state of affairs, are supported by suitable oracles. As for the "isomorphism conjecture" (recall 14.5.4), it is known that it fails under almost all oracles

- S. A. Kurtz, S. R. Mahaney, and J. S. Royer "The isomorphism conjecture fails relative to a random oracle," *Proc. 21st ACM Symp. on the Theory of Computing*, pp. 157–166, 1989,

while it was recently shown that it does hold for some:

- S. Fenner, L. Fortnow, and S. A. Kurtz "An oracle relative to which the isomorphism conjecture holds," *Proc. 33rd IEEE Symp. on the Foundations of Computer Science*, pp. 29–37, 1992.

14.5.8 It is quite challenging to give the right definition of space-bounded oracle computations. The difficulty is this: Should the query string be included in accounting for space, or should it be treated as an output string? For a discussion of the issue see

- J. Hartmanis "The structural complexity column: Some observations about relativization of space-bounded computations," *Bull. EATCS 35*, pp. 82–92, 1988.

Incidentally, this is one of a series of excellent commentaries by Juris Hartmanis on various aspects of complexity, starting in the 31st volume of the Bulletin.

14.5.9 Problem: Show that L has polynomial circuits if and only if $L \in \mathbf{P}^A$ for some *sparse* language A. (A non-uniform family of circuits is very much like a sparse oracle, containing polynomial information for each input length.)

14.5.10 Problem: Define a *robust* oracle machine $M^?$ deciding language L to be one such that $L(M^A) = L$ for all oracles A. That is, the answers are always correct, independently of the oracle (although the number of steps may vary from oracle to oracle). If furthermore M^A works in polynomial time, we say that oracle A *helps* the robust machine $M^?$. Let $\mathbf{P}_{\text{h}}$ be the class of languages decidable in polynomial time by deterministic robust oracle machines that can be helped; and $\mathbf{NP}_{\text{h}}$ for nondeterministic machines.

(a) Show that $\mathbf{P}_{\text{h}} = \mathbf{NP} \cap \mathbf{coNP}$.

(b) Show that $\mathbf{NP}_{\text{h}} = \mathbf{NP}$. (These concepts and results are from

- U. Schöning "Robust algorithms: A different approach to oracles," in *Theoretical Computer Science*, 40, pp. 57–66, 1985.)

14.5.11 For a long time the best known lower bound for the monotone circuit complexity of any monotone function had been *linear* (as is currently the case for the

non-monotone circuit complexity). In a fantastic breakthrough, Razborov proved in 1985 a *superpolynomial* (not yet exponential as in Theorem 14.6) lower bound for CLIQUE in

- A. A. Razborov "Lower bounds on the monotone complexity of some Boolean functions," *Dokl. Akad. Nauk SSSR, 281*, 4, pp. 798–801, 1985. English translation in *Soviet Math. Dokl., 31*, pp. 354–357, 1985.

By making better use of Razborov's technique the bound was soon improved to a true exponential:

- A. E. Andreev "On a method for obtaining lower bounds for the complexity of individual monotone functions," *Dokl. Akad. Nauk SSSR, 282*, 5, pp. 1033–1037, 1985. English translation in *Soviet Math. Dokl., 31*, pp. 530-534, 1985, and
- N. Alon and R. B. Boppana "The monotone circuit complexity of Boolean functions," *Combinatorica, 7*, 1, pp. 1–22, 1987.

Our exposition follows that in

- R. B. Boppana and M. Sipser "The complexity of finite functions," pp. 758–804 in *The Handbook of Theoretical Computer Science, vol. I: Algorithms and Complexity*, edited by J. van Leeuwen, MIT Press, Cambridge, Massachusetts, 1990.

For a short while it had been thought that perhaps Razborov's technique could, appropriately extended, establish Conjecture B (recall Section 11.4), and $\mathbf{P} \neq \mathbf{NP}$. For example, for all we knew then, any circuit that computes a monotone function has an equivalent monotone circuit of polynomially related size, and thus CLIQUE has no polynomial circuits, monotone or not. Such hopes were frustrated by Razborov himself, who showed that even polynomial problems such as MATCHING (Section 1.3) can have superpolynomial monotone complexity:

- A. A. Razborov "A lower bound on the monotone network complexity of the logical permanent," *Mat. Zametky, 37*, 6, pp. 887–900, 1985. English translation in *Russian Math. Notes, 37*, pp. 485–493, 1985.

Thus, NOT gates can be exponentially economical in expressing Boolean functions, and there is no general method that transforms any circuit computing a monotone function to an equivalent monotone circuit of comparable size.

14.5.12 On the subject of important theorems that bear a superficial similarity to $\mathbf{P} \neq \mathbf{NP}$ we should also mention here the $\mathbf{TIME}(n) \neq \mathbf{NTIME}(n)$ result proved in

- W. J. Paul, N. Pippenger, E. Szemerédi, and W. T. Trotter "On determinism versus nondeterminism and related problems," in *Proc. 24th IEEE Symp. on the Foundations of Computer Science*, pp. 429–438, 1983.

This result, like $\mathbf{TIME}(n) \subseteq \mathbf{SPACE}(\frac{n}{\log n})$ (recall Problem 7.4.17), uses a block-respecting Turing machine M and the graph $G_M(x)$. The basic graph-theoretic fact (analogous to part (c) of that problem) is now the following: In any computation graph of a k-string machine with N nodes there is a set S of $\mathcal{O}(\frac{kN}{\log^* N})$ nodes such that for any node $v \notin S$ there are $\mathcal{O}(\frac{N}{\log^* N})$ nodes $u \notin S$ such that there is a path from u to v; here $\log^* N$ is a very slowly growing function, namely the number of

times that we have to take logarithms so that N is reduced to a number below one. Such a set S is called a *segregator*.

By guessing and using the segregator, a nondeterministic machine can simulate a deterministic one with a savings of $\log^* N$ time; however, a stronger form of nondeterminism is needed: *Alternation* between the existential mode of **NP** and the universal mode of **coNP**, see Chapters 16, 17, and 19. In fact, the machine that simulates M with a savings of $\log^* N$ time uses *four* such mode alternations, also counting the initial existential one; this establishes that $\mathbf{TIME}(n \log^* n)$ is contained in a class that we could call $\Sigma_4\mathbf{TIME}(n)$; the the latter class captures this "quadruple" nondeterminism explained above (see Section 17.2 for extensions of **P** along this direction).

Now for the final result: It is easy to see that, if $\mathbf{TIME}(n) = \mathbf{NTIME}(n)$, then also $\mathbf{TIME}(n \log^* n) = \Sigma_4\mathbf{TIME}(n \log^* n)$ (we prove such an implication in Theorem 17.9), which in turn must be contained in $\Sigma_4\mathbf{TIME}(n)$. This contradicts the very fine hierarchy of nondeterministic time (and of the stronger variant of nondeterministic time employed here, see the references in Chapter 7).

14.5.13 Linear programming and the TSP. An approach that has been traditionally very successful for attacking combinatorial optimization problems is by formulating them as *set of linear inequalities* (recall the discussion in 9.5.34) and thus solving them. For example, the TSP (D) can be formulated as a set of linear inequalities in the variables x_{ij}, with intended values 1 if the optimum tour goes from city i to city j, and 0 otherwise. It is not hard to see that such linear programs exist but, unfortunately, they must provably involve an exponential number of inequalities.

One possible remedy is to introduce a set of new variables $y_1, \ldots, y_N$ and express the TSP (D) as a set of linear inequalities in the x_{ij}'s and the y_k's. Presumably, this set of linear inequalities will be *symmetric*, that is, intuitively, it will be invariant under any permutation of the cities. If this can be done with a polynomial number of extra variables and inequalities, we will have shown that $\mathbf{P} = \mathbf{NP}$.

As a matter of fact, one such construction was proposed in 1986 in

- E. R. Swart "P=NP," Technical Report, University of Guelph, 1986; revised 1987.

As was to be expected, this paper created much excitement in the community, and its involved construction was checked by many researchers, until, unfortunately, an error was identified.

Soon after this, in a remarkable paper Mihalis Yannakakis proved that *there can be no symmetric linear program for the* TSP *with less than exponential size:*

- M. Yannakakis "Expressing combinatorial optimization problems by linear programs," *Proc. 20th ACM Symp. on the Theory of Computing*, pp. , 223–228, 1988; also, *J.CSS 43*, pp. 441–466, 1981.

There are some interesting connections and parallels with Razborov's theorem. First, Yannakakis points out that a general, non-symmetric polynomial linear program for a hard special case of TSP (D) (telling whether a graph has a Hamilton cycle, recall Theorem 9.7 and its corollary) exists if and only if **NP** has polynomial circuits. Therefore, in some sense symmetric programs are restrictions of circuits, an alternative to monotone circuits. Also, the technique can be extended to prove a similar exponen-

tial lower bound for the polynomial-time problem of *general, non-bipartite matching*, recall 1.4.14, just like Razborov's theorem extends to bipartite matching. (Nonbipartiteness here is required, because, as we know, there *is* a symmetric linear program of polynomial size for bipartite matching: The one that expresses it as a maximum flow problem, which in turn is easily expressed in terms of linear inequalities, recall 9.5.14.)

Incidentally, Yannakakis' result is the only exponential lower bound we have for an **NP**-complete problem in a restricted model of computation within which there had been a serious, if ill-fated, attack at proving $\mathbf{P} = \mathbf{NP}$.

PART

IV INSIDE P

It is ironic but hardly surprising: As computing power was increasing rapidly over the course of the past 50 years, computer scientists were responding by repeatedly restricting, *rather than relaxing, what they considered "a satisfactory computational solution" of a problem. In the era before the digital computer, the prevailing notion was that of the recursive function. When computers became available in the 1950s, and it became apparent that not all recursive problems deserve to be considered "satisfactorily solved," very generous subclasses of recursive functions were considered: The stacks of exponentials in the Grzegorczyk hierarchy were popular at the time. As our computational power and ambition further swelled in the 1960s, and actual hard problems were attacked in earnest, the resulting frustrations led to the definition of polynomial-time computation—a paradigm whose influence cannot be missed in this book.*

The advent of the parallel computer, with its phalanx of processors, has caused a further shrinking of what we think as "satisfactorily solved." Polynomial time is not good enough any more, because not all fast sequential algorithms can be massively parallelized. We must search deeper into P—and our souls—to discover the concepts and paradigms appropriate for the new realities.

CHAPTER

15 PARALLEL COMPUTATION

Some of the least understood areas in science—be it in physics, economics, or computation—seem to involve the concurrent interaction of large numbers of agents.

15.1 PARALLEL ALGORITHMS

In the past 20 years, parallelism has been deeply transforming the theory and practice of computation—at first as a far-fetched futuristic possibility, currently as a most challenging reality. Parallel computers exist now that have enormously large number of processors, all of which cooperate for the solution of the same problem instance. To fix ideas, we shall be thinking of a parallel computer with a large number of independent processors. Each processor can execute its own program, and can communicate with other processors instantaneously and synchronously through a large shared memory. That is, all processors execute their first instruction in unison, then they exchange information, then they execute the second instruction, and so on. This kind of multiprocessor is not the only one, but it is the easiest about which to think and for which to write algorithms. (On the negative side, it is also the hardest kind to build and scale up to truly large numbers of processors.)

In designing algorithms for such machines we obviously want to minimize the time between the beginning and the end of the concurrent computation—because this is precisely the point of building parallel computers. In particular, we would like our parallel algorithms to be *dramatically faster* than our sequential ones, and we shall see how we can formalize this goal. Naturally, our algorithms should not require inordinately large numbers of processors.

But the best way to understand parallelism—its power, its intricacies, and its limitations—is to describe first informally a few diverse examples of parallel algorithms, just as we did in Chapter 1 for sequential algorithms.

Matrix Multiplication

The design of parallel algorithms is usually a much harder undertaking than is the design of sequential algorithms, since so many more issues and parameters are involved. Once in a while, however, the situation is quite simple: The straightforward sequential algorithm can be *readily parallelized.*

A good example of this phenomenon is *matrix multiplication.* Suppose that we are given two $n \times n$ matrices A and B, and wish to compute their product $C = A \cdot B$; that is, we wish to compute all n^2 sums of the form

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}, \quad i, j = 1, \ldots, n.$$

Sequentially this problem can be solved in $\mathcal{O}(n^3)$ arithmetic operations in the obvious way (there are clever and sophisticated algorithms for this problem that are asymptotically faster than n^3, but let us disregard this point here).

We can obtain a satisfactory parallel algorithm for this problem simply *by extracting as much parallelism as possible* from the sequential one. The n^3 products $A_{ik} \cdot B_{kj}$ can be computed independently and *by different processors*—we are assuming here that we have n^3 processors. We call the processor that computes the $A_{ik} \cdot B_{kj}$ product *the* (i, k, j) *processor.* Then, n^2 of these processors, say the $(i, 1, j)$ ones, can each collect the n products corresponding to the same C_{ij} (here we use our assumption that communication between processors is instantaneous), and add them up in $n - 1$ additional steps.

The total time required is n arithmetic operations on n^3 processors. Bringing down the complexity from n^3 to n is of course significant, but it is not the kind of dramatic improvement that would make multiprocessors worth building. What we would like to see is some *exponential drop* in the time required, say to parallel time $\log n$ (or at least *polylogarithmic* parallel time, like $\log^2 n$ or $\log^3 n$).

And there is a simple way to accomplish this: Instead of assigning all additions to the same processor, we can organize the processors to perform a binary tree of additions. This way, we can add the results in only $\log n$ parallel steps. In more detail, at the sth step, processor $(i, 2^s \lfloor \frac{k}{2^s} \rfloor, j)$ computes the sum of its contents and the contents of processor $(i, 2^s \lfloor \frac{k}{2^s} \rfloor + 2^s, j)$, where s ranges from zero to $\lceil \log n \rceil - 1$. In the end, processor $(i, 1, j)$ will contain the result C_{ij}, as before. The total number of parallel steps is thus $\log n + 1$, and the number of processors used is n^3.

Our goal in parallel algorithms is to achieve such logarithmic, or at least polylogarithmic, such as $\log^3 n$, *step counts.* This is the exponential drop in complexity that we hope to obtain from parallel computers—analogous to breaking the barrier between exponential and polynomial time for sequential computation. It is also important that the number of processors is a polynomial—because an exponential number of processors is even less feasible than exponential sequential time.

Naturally, the "steps" needed in this example are arithmetic operations on the entries of the matrix. If these entries are integers, we must think how such operations on long integers can be broken down into bit operations and executed in parallel (this is done later in this section). But we shall soon see that the matrix multiplication problem is rather interesting even in the case of *Boolean matrices*, for which special case our step count is accurate.

Can we do better? It is not hard to see that $\log n$ parallel steps are necessary for multiplying matrices, under the broadest assumptions and models. How about the number of processors needed? Since we have decided to consider implementations of the obvious $\mathcal{O}(n^3)$ sequential algorithm for matrix multiplication, the *amount of work* done by our algorithm must be at least as large (by "amount of work" we mean the steps executed by each processor, summed over all processors). The reason is a general principle that is as valuable as it is obvious: *The amount of work done by a parallel algorithm can be no smaller than the time complexity of the best sequential algorithm* (or the best we want to consider—in this case, the n^3 one). Any parallel algorithm can be simulated by a sequential one that does the same amount of work.

Now, it is obvious that any algorithm that does work at least n^3 and achieves the optimum parallel time $\log n$ *requires at least* $\frac{n^3}{\log n}$ *processors.* The question is, can we decrease our processor requirement from n^3 to the optimum $\frac{n^3}{\log n}$ without increasing the parallel time too much?

Here is how: We compute the n^3 products not in a single step as before, but rather in $\log n$ "shifts" using $\lceil \frac{n^3}{\log n} \rceil$ processors at each shift. We use shifts of the same $\lceil \frac{n^3}{\log n} \rceil$ processors to compute the first $\log \log n$ parallel addition steps, where more than $\frac{n^3}{\log n}$ processors would be ordinarily needed. The total number of parallel steps is now no more than $2 \log n$, with $\frac{n^3}{\log n}$ processors, thus rendering our parallelization of the $\mathcal{O}(n^3)$ sequential algorithm optimal in all respects—give or take a factor of 2. This important technique of bringing down the processor requirement to the optimum value (for given work and parallel time) by using "shifts of processors" is quite general and valuable; it is known as *Brent's principle.*

Expressing processor requirements as a function of n may seem bizarre. After all, any parallel machine has a given and fixed number of processors, and

all kinds and sizes of instances will have to be solved on it. For example, what if we have to multiply $n \times n$ matrices in a parallel computer with P processors, where P is much smaller than $\frac{n^3}{\log n}$? Once we have an algorithm that achieves optimal parallel time using as many processors as it takes, we can now scale back our algorithm to the available hardware. In this case, we can organize our P processors so that they execute each parallel step of our algorithm in $\lceil \frac{n^3/\log n}{P} \rceil$ shifts, where each shift employs P processors. The total time is $\frac{2n^3}{P}$, which is obviously the fastest that this algorithm can be parallelized on P processors.

Incidentally, this is another aspect of the reason why we are so eager to minimize the number of processors in our parallel algorithms: Using too many processors in our basic algorithm eventually translates to high parallel time when we apply our algorithm to large instances on a machine with a fixed number of processors.

Graph Reachability

Let us next examine the REACHABILITY problem, so fundamental in sequential computation, from the standpoint of parallelism. This problem exemplifies the sobering truth about parallel algorithms: To develop a parallel algorithm for a problem, we often have to *forget all we know about sequential algorithms* for the problem, and to start from scratch, with completely different ideas. We shall see this pattern again and again in the examples of this section.

The search algorithm for REACHABILITY (Section 1.1) cannot be parallelized in any obvious way: Even if we cleverly arrange that many processors get nodes from the stack (or queue) S and process the nodes simultaneously so that no chaos ensues, still the number of parallel steps will be *at least equal to the shortest path from the start node to the goal node*—and this path can be as long as $n-1$, say if the graph is simply a path. In fact, it is suspected that performing depth-first search is one of those problems that are inherently sequential, those that cannot be parallelized in polylogarithmic time. So, we should not pursue the search idea, and we should look for a completely different approach.

One interesting approach uses the problem we were so successful in solving: Matrix multiplication. Suppose that A is the adjacency matrix of the graph, where we have added all self-loops: $A_{ii} = 1$ for all i. Suppose now that we compute the Boolean product of A with itself $A^2 = A \cdot A$, where

$$A^2_{ij} = \bigvee_{k=1}^{n} A_{ik} \wedge A_{kj}.$$

A little reflection shows that $A^2_{ij} = 1$ *if and only if there is a path of length 2 or less from node i to node j.*

Why stop? Computing $A^4 = A^2 \cdot A^2$, we get all paths of length 4 or less, then with A^8 the paths of length 8 or less, and so on. After $\lceil \log n \rceil$

Boolean matrix multiplications, we get $A^{2^{\lceil \log n \rceil}}$, which is *the adjacency matrix of the transitive closure of* A—and the transitive closure is nothing else but the concentrated answers to all possible REACHABILITY instances on the given graph. It follows that the transitive closure of a graph can be computed in $\mathcal{O}(\log^2 n)$ parallel steps with $\mathcal{O}(n^3 \log n)$ total work. Notice that this result is exactly the kind that we have been seeking: The parallel time is polylogarithmic, and the amount of total work is polynomial (and, by Brent's principle, so is the number of processors required, $\mathcal{O}(\frac{n^3}{\log n})$). If we need to answer just one REACHABILITY question, then only $\mathcal{O}(n^2 \log n)$ work is needed.

Arithmetic Operations

Suppose that we are given n integers $x_1, \ldots, x_n$, and we wish to compute all sums of the form $\sum_{i=2}^{j} x_i, \quad j = 1, \ldots, n$. This problem, known as *the prefix sums problem*, is trivial to solve sequentially with $n-1$ additions—we just compute $x_1 + x_2$, from this $x_1 + x_2 + x_3$, and so on up to $x_1 + x_2 + \ldots + x_n$. Unfortunately, this algorithm is very sequential, inappropriate for parallelization.

Our parallel algorithm for prefix sums is best described recursively. Assume that n is a power of 2—otherwise, pad the sequence with enough innocuous elements. To compute the prefix sums of $x_1, \ldots, x_n$ we first compute the sums $(x_1 + x_2), (x_3 + x_4), \ldots, (x_{n-1} + x_n)$ (one parallel step). We then recursively compute the prefix sums of *this* sequence. At this point we have one half of the answers we need—namely those for all even positions. Each of the remaining $\frac{n}{2}$ answers can be computed from these and the original inputs with one more parallel addition step. Thus, computing prefix sums of sequences of length n involves two more parallel steps (one in the beginning and one in the end) than does computing prefix sums of sequences of length $\frac{n}{2}$. It follows that the total number of parallel steps is $2 \log n$ -there is no faster algorithm, even if we want to compute just $\sum_{i=1}^{n} x_i$. The amount of work needed is $n + \frac{n}{2} + \frac{n}{4} + \ldots \leq 2n$ and thus, by Brent's principle, the number of processors needed is only $\frac{n}{\log n}$.

In fact, it is not hard to argue that a much more general problem has been solved: The operation "+" in our definition of prefix sums need not be integer addition, but *any associative operation* on any domain, as long as $a + (b + c) = (a+b)+c$. We next use this generalized prefix sums idea in our parallel algorithm for *binary addition*.

Long addition of two n-bit binary numbers is yet another example of an operation that is easy sequentially (the elementary-school algorithm takes $\mathcal{O}(n)$ Boolean operations), but is tricky to parallelize. We are given two binary numbers $a = \sum_{i=0}^{n} a_i 2^i$ and $b = \sum_{i=0}^{n} b_i 2^i$, and wish to find their sum $c = \sum_{i=0}^{n} c_i 2^i$, where all a_is, b_is, and c_is are $0-1$, and $a_n = b_n = 0$. To compute c_i we have to compute the *carry* z_i out of the ith position; if we have computed

all these carries, then $c_i = a_i + b_i + z_{i-1} \bmod 2$, and we can compute all bits of the result in two more parallel steps. The problem is that carry tends to *propagate*, so that computing the ith carry seemingly requires that the $i-1$st carry has been computed first.

What is the formula for z_i? By definition, the carry z_i is 1 if either (a) both a_i and b_i are 1, or (b) if at least one of them is 1 and *the previous carry is* 1. That is, if we define $g_i = a_i \wedge b_i$ to be *the carry generate bit* at the ith position, and $p_i = a_i \vee b_i$ *the carry propagate bit*, then we can write

$$z_i = g_i \vee (p_i \wedge z_{i-1}). \tag{1}$$

To start off, we assume that $z_{-1} = 0$.

Substituting in equation (1) the formula for z_{i-1} we get

$$z_i = [g_i \vee (p_i \wedge g_{i-1})] \vee ([p_i \wedge p_{i-1}] \wedge z_{i-2})\,. \tag{2}$$

Notice that recurrence (2), seen as a formula for obtaining z_i from z_{i-2}, is the same kind of recurrence as (1), except that g_i has been replaced by $[g_i \vee (p_i \wedge g_{i-1})]$, and p_i by $[p_i \wedge p_{i-1}]$. That is, we can think of computing z_i by back-substituting in (1) as a special kind of sum of the bit vectors $((0,0) \odot ((g_1,p_1) \odot \ldots \odot ((g_{i-1},p_{i-1}) \odot (g_i,p_i))\ldots))$, where the operation $\odot$ between bit vectors of length 2 is defined as follows:

$$(a,b) \odot (a',b') = (a' \vee (b' \wedge a), b' \wedge b).$$

Now *it is not hard to check that* $\odot$ *is an associative operation*. Therefore, computing all carry bits is the same as computing the "generalized prefix sums" of the bit vectors $(0,0), (g_1,p_1), (g_2,p_2), \ldots, (g_n,p_n)$ under the operation $\odot$; the previous algorithm can be applied to compute all carries z_i in $2\log n$ parallel $\odot$ steps, or, since $\odot$ takes three elementary operations, $6\log n$ parallel Boolean operations, and $\mathcal{O}(n)$ total work. Computing the final result, the sum of the two given n-bit integers, requires just two more parallel steps. We conclude that we can compute the sum of two n-bit binary integers in $\mathcal{O}(\log n)$ parallel time, and $\mathcal{O}(n)$ work.

This brings us to *multiplication* surely an even harder problem to solve in parallel. Suppose that we wish to multiply two n-bit binary integers — say $a \times b = 1001101110 \times 1011010011$. But this is exactly like adding together at most n integers, namely $1001101110 + 10011011100 + 1001101110000 + 100110111000000 + 1001101110000000 + 100110111000000000$. Each of these numbers is a, multiplied by a power of 2 for which the corresponding bit of b is one. It is not hard to see that these numbers can be computed from a and b in logarithmic parallel time and $\mathcal{O}(n^2)$ hardware (one way is to generate all numbers $a, 2a, 4a, \ldots, 2^n a$ by prefix sums, and then "mask out" in parallel

the ones that correspond to zero bits of b). Thus, to multiply n-bit integers in parallel, it suffices to show how to add n or fewer $2n$-bit integers in parallel.

we can do this by the so-called *two for three* trick: Suppose that a, b, c are three of the $2n$-bit integers to be added, and let a_i, b_i, c_i denote their ith bit. If we add these three bits, we get a two-bit number $a_i + b_i + c_i = 2 \cdot p_i + q_i$, where p_i and q_i are also bits. Now the p_is and q_is spell two other n-bit numbers, call them p and q, and so $a + b + c = 2 \cdot p + q$. Thus, in one step, we have reduced adding three integers to adding two integers—q, and p with a zero added to its end.

Our algorithm for adding n or fewer $2n$-bit integers is exactly this: We subdivide them into triples (ignoring any remaining integers), and in one parallel step replace each of these triples by two $2n + 1$-bit integers. That is, at each parallel step the number of integers is multiplied by $\frac{2}{3}$. After at most $\log_{\frac{3}{2}} 2n$ steps, we have one or two integers, which we add. It follows that multiplication can be performed in $\mathcal{O}(\log n)$ parallel time, with $\mathcal{O}(n^2 \log n)$ work. Faster asymptotic methods are known (see the references).

Maximum Flow

The limitations of parallelism are best exemplified by the MAX FLOW problem. The crucial weakness, with respect to parallel computation, of our sequential algorithm for finding the maximum flow in a network N (Section 1.2) is that it works *in stages*. At each stage, we start with a flow f (at the first stage we have the everywhere-zero flow), and try to improve it. To this end, we construct a new network $N(f)$, reflecting the improvement potential of the arcs of N with respect to f, and try to find a path from the source s to the sink t in $N(f)$. If we succeed, we improve the flow. If we fail, the current flow is maximum.

It is not hard to see that each stage can be parallelized most satisfactorily. With enough hardware, we can construct $N(f)$ in a single parallel step —and we know how to find paths quickly in parallel from earlier in this section. Thus, each stage can be done in $\mathcal{O}(\log^2 n)$ parallel time and $\mathcal{O}(n^2)$ total work, where n is the number of nodes in the network. *The problem is that stages need to be carried out one after the other,* and the number of stages may be very large—certainly more than polylogarithmic in n.

As always, we may try to develop alternative approaches to the problem that are more susceptible to parallelization than is the obvious sequential one. For example, we may try to merge consecutive stages so that they can all be performed in parallel at the same time—we could try finding more than one augmenting path at a time, for example all possible augmenting paths of the same length. Indeed, such stratagems may succeed in reducing the number of stages to n, or even $\sqrt{n}$ in some cases, but not to a polylogarithmic number. MAX FLOW remains a prime example of a polynomial-time solvable problem that seems to be *inherently sequential.* In a later section we shall prove formally

that, in some rigorous sense, it is.

The Traveling Salesman Problem

Our discussion of parallel algorithms so far has made another important point clear: Parallel computation is *not*, as we may have naively hoped, the answer to **NP**-completeness; it is not the technological breakthrough that will render exponential algorithms feasible. The obstacle is the equation

$$\text{work} = \text{parallel time} \times \text{number of processors}.$$

If the fastest sequential algorithm that we know for a problem requires exponential time (as is currently the case for all **NP**-complete problems), then in any parallel algorithm either the parallel time must be exponential, or the number of processors must be exponential (or both). It is unlikely that humankind will ever construct machines with truly astronomically many processors—the limitations here are even more severe than are the limits of our patience.

We do not imply here that parallel computers are useless in attacking hard problems. Parallel computation, along with clever exponential algorithms, fast processors, and smart programming techniques, do help solve exactly larger and larger instances of **NP**-complete problems. The point is that the specter of **NP**-completeness and exponentiality cannot be exorcised by parallelism alone.

Determinants and Inverses

We conclude this section with a fundamental problem which also appears at first glance to be inherently sequential, but for which a rather sophisticated approach succeeds in providing a fast parallel algorithm: The problem is that of computing the determinant of an integer matrix.

We have already seen in Section 11.1 that a determinant can be computed in $\mathcal{O}(n^3)$ arithmetic operations by the method of Gaussian elimination. Now, Gaussian elimination is a very sequential algorithm, as it must process one row of the matrix after the other. Of course, each processing of a row can be parallelized easily, but still the number of parallel steps needed would be n or worse.

An alternative approach solves this problem by merging it with another difficult problem, *matrix inversion*, and then solving both. Let us explain the connection between determinants and inversion. Suppose that A is a matrix; let $A[i]$ denote the matrix resulting if we omit the first $n - i$ rows and columns of A—that is, $A[i]$ is the $i \times i$ lower right-hand corner of A. Consider the inverse $A[i]^{-1}$ of this matrix, and its first element $(A[i]^{-1})_{11}$. Cramer's rule says that

$$(A[i]^{-1})_{11} = \frac{\det A[i-1]}{\det A[i]},$$

and this holds for $i = n, n-1, \ldots, 2$. Back-solving these equations, and since $A[n] = A$, we obtain

$$\det A = (\prod_{i=1}^{n} (A[i]^{-1})_{11})^{-1}. \tag{3}$$

We shall use this improbable formula for computing determinants in parallel. That is, we shall compute determinants by first computing the inverses of many matrices, all in parallel, then multiplying the upper-left entries, and finally inverting the result.

But there is yet another complication: We cannot apply this directly to the original matrix A, but to a *symbolic* matrix derived from A. Naturally, we know from Section 11.1 that computing symbolic determinants is asking for trouble. Fortunately, our symbolic matrices will have only one variable: We shall compute the determinant of the matrix $I - xA$, using (3). And inverting matrices of the form $I - xA$ turns out to be easy.

The inspiration for computing $(I - xA)^{-1}$ comes from considering the same problem for 1×1 matrices A, that is, for real numbers: The formal power series is

$$(1 - xA)^{-1} = \sum_{i=0}^{\infty} (xA)^i. \tag{4}$$

Thus in order to compute $(I - xA[i])^{-1}$ we just have to compute and add in parallel powers of $xA[i]$ (using prefix sums).

But how does one deal with the worrisome infinite summation in (4)? Let us recall that we only need to compute the determinant of $I - xA$, and this determinant is a polynomial in x of degree n. Thus, we can carry out the computation in (4), and beyond, *truncating the power series after the x^n term.* That is, all our computations will involve matrices of polynomials of degree n. As a consequence, the summation in (4) is stopped at the nth addend. We thus compute in parallel all $(I - xA)[i]^{-1}$s, each by computing by parallel prefix all matrices of the form $(xA)^i \bmod x^{n+1}$ (where mod x^{n+1} reminds us to ignore terms higher than x^n), and then adding them together.

Once we have all $(I - xA)[i]^{-1}$s, we obtain their upper-left elements and multiply them together modulo x^{n+1} to obtain a polynomial of degree n in x, call it $c_0(1 + xp(x))$ for some number $c_0 \neq 0$ (if it so happens that the constant coefficient of this polynomial is zero, the calculation that follows can be easily modified). This polynomial is, by virtue of (3), precisely the inverse of $\det(I - xA)$. We can then compute the determinant by inverting this polynomial, again using the power series for inversion, appropriately truncated after the x^n term:

$$(c_0(1 + p(x)))^{-1} = \frac{1}{c_0} \sum_{i=1}^{\infty} (-xp(x))^i \bmod x^{n+1}.$$

We have thus computed $\det(I - xA)$. Of course we are interested in computing $\det A$; but this can now be obtained easily as *the coefficient of* x^n *in* $\det(I - xA)$, multiplied by -1 if n is odd (in proof, consider the limit of $\frac{1}{(-x)^n}\det(I - xA)$ when x goes to infinity). This concludes our description of the parallel algorithm for computing determinants.

Example 15.1: Suppose that we wish to compute the determinant of

$$A = \begin{pmatrix} 1 & -2 \\ -1 & 3 \end{pmatrix}$$

using this method. We start with

$$I - xA = \begin{pmatrix} 1-x & -2x \\ x & 1-3x \end{pmatrix},$$

and we must compute the $(I - xA)[i]_{11}^{-1}$s for $i = 1, 2$.

The case $i = 1$ is always easy. Matrix $xA[1]$ is just $(3x)$, and therefore $\sum_{i=0}^{\infty}(xA[1])^i \bmod x^3 = (1 + 3x + 9x^2)$. The upper-left element of this matrix is, of course, $1 + 3x + 9x^2$.

To compute $(I - xA)[2]^{-1}$, we need the powers

$$(xA[2])^0 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad (xA[2])^1 = \begin{pmatrix} x & 2x \\ -x & 3x \end{pmatrix}, \quad (xA[2])^2 = \begin{pmatrix} -x^2 & 8x^2 \\ -4x^2 & 7x^2 \end{pmatrix};$$

all higher powers will be ignored since we are working modulo x^3. Adding those together we get that

$$(I - xA)[2]^{-1} = \begin{pmatrix} 1 + x - x^2 & 2x + 8x^2 \\ -x - 4x^2 & 1 + 3x + 7x^2 \end{pmatrix} \bmod x^3,$$

and thus $((I - xA)[2]^{-1})_{11} = 1 + x - x^2$. Multiplying $((I - xA)[1]^{-1})_{11}$ times $((I - xA)[2]^{-1})_{11}$ we get

$$(1 + 3x + 9x^2)(1 + x - x^2) = 1 + 4x + 11x^2 = 1 + x(4 + 11x) \bmod x^3.$$

We must now invert this polynomial modulo x^3; we have to calculate

$$1 - (4x + 11x^2) + (4x + 11x^2)^2 = 1 - 4x + 5x^2 \bmod x^3,$$

from which we can read the value of the determinant of A as the coefficient of x^2: The answer is 5 (as we had known all along...). □

Each of the three stages of this complicated algorithm (computing the inverses, multiplying corner elements, inverting the result) can be carried out in

$\mathcal{O}(\log^2 n)$ parallel steps. The amount of work needed is awesome, but polynomial: The first stage is the most demanding, and it needs n parallel matrix multiplications, or $\mathcal{O}(n^4)$ total work. Unfortunately, the matrix elements are not bits, but nth degree polynomials in x. It is easy to see that each arithmetic operation on polynomials can be done in $\mathcal{O}(\log n)$ parallel arithmetic steps, for $\mathcal{O}(n^2)$ total work.

We are not done yet: If the elements of the original matrix A are b-bit integers, then it is not hard to see that the coefficients of these polynomials have $\mathcal{O}(nb)$ bits, and each arithmetic operation takes $\mathcal{O}(\log n + \log b)$ bit operations and $\mathcal{O}(n^2 b^2)$ total work. We conclude that we can compute the determinant of an $n \times n$ matrix with b-bit integer entries in parallel time $\mathcal{O}(\log^3 n(\log n + \log b))$, and $\mathcal{O}(n^8 b^2)$ total work. Although unrealistically large, these bounds still conform to our theoretical requirements of polylogarithmic parallel time and polynomial work in the size of the input (which is $n^2 b$).

15.2 PARALLEL MODELS OF COMPUTATION

In Chapter 2 we introduced several related models of computation: The Turing machine, its multistring variant, the RAM, and the nondeterministic Turing machine. The first three were unambiguously *sequential:* Their most fundamental characteristic was the so-called *von Neumann property*, namely that at each instant only a bounded amount of computational activity can occur. Nondeterminism exhibits some of the attributes of parallel computation, if one considers each level of the computation tree (recall Figure 2.9) as unbounded concurrent activity. But it is a weak form of parallelism, in that the various processes can only communicate at the end, via a restricted form of "consensus voting;" we shall see in the next chapter a generalization of nondeterminism which captures parallelism most faithfully.

But we have already seen a model that is genuinely parallel: *The Boolean circuits* introduced in Section 4.3, and further studied in Sections 11.3 and 14.4. A Boolean circuit has no "program counter," and thus its computational activity may take place at many gates concurrently.

Boolean circuits will be our basic model of parallel algorithms. Since we are interested in solving instances of arbitrary size, we consider *families* of Boolean circuits with one different circuit for each input size, as introduced in Section 11.3. Recall that a *circuit family* is a sequence $\mathcal{C} = (C_0, C_1, \ldots)$ of Boolean circuits, such that C_i has i inputs. To avoid absurd families like the one that solves an undecidable problem constructed in the proof of Proposition 11.6, we only consider families of circuits that are *uniform.* That is, there is a logarithmic space-bounded Turing machine which on input 1^n outputs C_n; intuitively, this implies that all circuits in the family are related to the same algorithmic idea, that they represent the same algorithm.

Example 15.2: We saw in Example 11.3 a uniform family of polynomial circuits, called $\mathcal{C}_1$, which solve the REACHABILITY problem. The nth circuit in that family had size $\mathcal{O}(n^3)$, and was based on a recursion for computing the transitive closure of a graph.

In the previous section we saw an algorithm for computing the transitive closure (and therefore for solving REACHABILITY) that is far more suited for parallel computation: The repeated squaring of the adjacency matrix $\log n$ times. This algorithm can be easily rendered as a uniform family of circuits as follows: First, consider, for each n, a circuit Q with n^2 inputs and n^2 outputs, and such that the output Boolean matrix is the square of the input one. Now the circuit for transitive closure is simply the *composition* of $\lceil \log n \rceil$ copies of Q, connected *in tandem* so that the outputs of one coincide with the inputs of the next. We call the resulting family $\mathcal{C}_2$. □

Once we defined Turing machines in Chapter 2, we immediately proceeded to define the time and space required by their computation. In parallel computation we have two important new complexity measures: *Parallel time* and *work*.

Definition 15.1: Let C be a Boolean circuit, that is, a directed acyclic graph where each node is a gate, of one of the possible sorts and matching indegree. (C could have more than one output, in which case it computes a function from $\{0,1\}^n$ to $\{0,1\}^m$, not a predicate.) The *size* of C is, as always, the total number of gates in it. The *depth* of C is *the number of nodes in the longest path in* C.

Let now $\mathcal{C} = (C_0, C_1, \ldots)$ be a uniform family of circuits, and let $f(n)$ and $g(n)$ be functions from the integers to the integers. We say that the *parallel time* of $\mathcal{C}$ is at most $f(n)$ if for all n the depth of C_n is at most $f(n)$. We say that the *total work* of $\mathcal{C}$ is at most $g(n)$ if for all $n \geq 0$ the size of C_n is at most $g(n)$.

Finally define $\mathbf{PT/WK}(f(n), g(n))$ to be the class of all languages $L \subseteq \{0,1\}^*$ such that there is a uniform family of circuits $\mathcal{C}$ deciding L with $\mathcal{O}(f(n))$ parallel time and $\mathcal{O}(g(n))$ work. Notice that, in the absence of a "linear speedup theorem" for parallel time and work (analogous to Theorem 2.2 for sequential time) our definition of parallel complexity classes explicitly disregards multiplicative constants, via the $\mathcal{O}(\cdot)$ notation used. □

Example 15.2 (Continued): The uniform family of circuits $\mathcal{C}_1$ for REACHABILITY shows that REACHABILITY$\in \mathbf{PT/WK}(n, n^3)$ —the depth of those circuits was n, because of the recursion. On the other hand, the family $\mathcal{C}_2$ establishes that REACHABILITY$\in \mathbf{PT/WK}(\log^2 n, n^3 \log n)$. □

Notice that, while in sequential computation our study of time and space was quite disjointed, in parallel computation we simultaneously bound *both* parallel time and work. One of the reasons is that, as we have seen in the

previous section when studying the parallel complexity of matrix multiplication, work and parallel time are interrelated, and in some sense interchangeable, as high work requirements may turn into long delays in an implementation with few processors.

But there is a deeper reason for such meticulousness, probably already evident to the reader: Exactly because the theory of parallel computation is such a young and insecure field, motivated and driven by a very current and important technology, it tends to be much more careful and conservative in style —less cavalier in its treatment of constants and exponents, more realistic in its models, less likely to focus on one issue at a time and forget the others.

Parallel Random Access Machines

As a model of computation, the Turing machine is every bit as clumsy and awkward to program as circuits. In Chapter 2 we boosted our confidence in Turing machines as a universal model of computation by proving that they can simulate without substantial loss of efficiency truly realistic models such as the RAM (recall Section 2.6).

We shall now engage in a similar maneuver. We shall define a parallel version of the RAM, one that models parallel computers quite accurately and convincingly, and then show that its power *vis à vis* parallel computation is closely related to that of circuits.

Definition 15.2: Recall from Section 2.6 that a RAM program is a finite sequence $\Pi = (\pi_1, \ldots, \pi_m)$ of instructions of the kinds shown in Figure 2.6 (READ, ADD, LOAD, JUMP, etc.), with arguments standing for the contents of registers (memory locations). Register 0 is the *accumulator* of the RAM, where the result of the current operation is stored. At each step, the RAM executes the instruction pointed by the program counter κ, reading and writing integer values on the registers as required by the instruction. There is also a set of input registers $I = (i_1, \ldots, i_m)$.

We now generalize this to a *PRAM program* (for *parallel random access machine*). A PRAM program is a sequence of RAM programs, $P = (\Pi_1, \Pi_2, \ldots, \Pi_q)$, one for each of q RAMS. We assume that each of these machines executes its own program, has its own program counter, its own accumulator (the accumulator of RAM i is Register i) but they all share—can both read and write—all registers. We assume in fact that each RAM can also read and write the accumulators of the other RAMs. There is no Register 0.

The number q of RAMs in the PRAM program is intended not to be a constant, but a function $q(m, n)$ of the number m of integers in the input I, as well as of the total length of these integers, denoted $n = \ell(I)$. In fact, the text of the programs is itself dependent on m and n. In other words, for each value of m and n we have a different PRAM program $P_{m,n}$, each with a different number

of RAMs $q(m,n)$, comprising a *two-dimensional family* $\mathcal{P} = (P_{mn} : m,n \geq 0)$ *of PRAM programs*. So that we avoid absurd cases, we shall only consider families of PRAM programs (*PRAMs* for short) that are *uniform*. That is, there is a Turing machine which, given $1^m 0 1^n$, generates the number $q(m,n)$ of processors, as well as the programs $P_{m,n} = (\Pi_{m,n,0}, \Pi_{m,n,1}, \ldots, \Pi_{m,n,q(m,n)})$, all in logarithmic space.

Notice that we allow the number of processors in a PRAM to depend both on the number of input integers, and their total length. The reason is that, depending on the problem, a PRAM may need more parallelism when the integers are large. Usually, $q(m,n)$ will depend only on m.

A *configuration* of the PRAM $P_{m,n}$ is a tuple $(\kappa_1, \kappa_2, \ldots, \kappa_{q(m,n)}, R)$, where the configuration now contains the program counters of all RAMs, together with R, a description of the current contents of the registers (recall the corresponding definition for the RAM). It is straightforward to extend the *yields* relation to such configurations: In one step the ith RAM executes the instruction indicated by program counter κ_i, on arguments fetched from the registers as mandated by the instruction, or from its own accumulator, Register i. There is only one subtlety: Since we allow more than one RAM to read *and write* any register, we must determine what happens if more than one processors try to update the same register (either by a STORE instruction, or by an arithmetic instruction, if that register is the RAM's own accumulator). *We adopt the convention that the RAM with the smallest index prevails* and has its value written in the register (see the references for alternatives and how they compare with our convention).

Finally, suppose that F is a function mapping finite sequences of integers to finite sequences of integers (this allows $0-1$ values, corresponding to decision problems); let $\mathcal{P} = (P_{m,n} : m,n \geq 0)$, be a uniform family of PRAM programs; and let f and g be functions from positive integers to positive integers. We say that $\mathcal{P}$ *computes* F *in parallel time* f *with* g *processors* if for each $m,n \geq 0$ $P_{m,n}$ has the following property: First, it has $q(m,n) \leq g(n)$ processors. Second, if the PRAM program is executed on input $I = (i_1, \ldots, i_m)$ of m integers with total number of bits $\ell(I) = n$, then all $q(m,n)$ RAMs have reached a HALT instruction after at most $f(n)$ steps, at which point the $k \leq q(m,n)$ first registers contain the output $F(i_1, \ldots, i_m) = (o_1, \ldots, o_k)$. □

Perhaps the definition of the PRAM is worth discussing a little. First, the PRAM is a remarkably faithful model of our "mental parallel machine," which we have programmed to solve so many diverse problems in the previous section. It is an extremely (and somewhat unrealistically) powerful parallel computer. Its processors are capable of instantaneous communication via their shared memory (usually only parallel machines with very few processors have such facility; when the number of processors P is large, communication is handled by a network, with an associated communication delay which is at best logarithmic

in P). In fact, PRAM processors even share *write* access to their common memory, something that is problematic to implement in hardware as an atomic step (naturally, we can always implement such a step by keeping one copy of all writes to each register, in order to sort out later which one prevails, again with logarithmic delay).

In other words, our PRAM is a most idealized and powerful model of parallel computation. In the light of this, its close relationship with circuits (a most primitive and realistic model) is reassuring. We next give results in both directions. First, it is hardly surprising that PRAMs can easily simulate circuits:

Theorem 15.1: If $L \subseteq \{0,1\}^*$ is in $\mathbf{PT/WK}(f(n), g(n))$, then there is a uniform PRAM that computes the corresponding function F_L mapping $\{0,1\}^*$ to $\{0,1\}$ in parallel time $\mathcal{O}(f(n))$ using $\mathcal{O}(\frac{g(n)}{f(n)})$ processors.

Proof: Using the logarithmic-space machine that generates the nth circuit C_n, we can generate equivalent RAM programs as follows (see the proof of Proposition 8.2 on how to compose two logarithmic-space machines into one). For each gate g_i of C_n we have a different RAM Π_i (we shall sketch later how to reduce the number of processors to $\mathcal{O}(\frac{g(n)}{f(n)})$).

The program of Π_i is very simple: First, it waits for $3d$ steps, where d is *the length of the longest path from any input gate to* g_i. This number is easy to calculate in logarithmic space from C_n. (Although strictly speaking our instruction repertoire for the RAM has no NOOP instruction, an instruction that does absolutely nothing, the reader can think of many ways to simulate one.) After this, Π_i in three steps computes the value of g_i and stores it in its accumulator, Register i. If g_i is an AND gate with inputs g_j and g_k, then Π_i executes the following RAM program:

$3d+1$. LOAD j
$3d+2$. JZERO $3d+5$
$3d+3$. LOAD k
$3d+4$. JUMP $3d+6$
$3d+5$. LOAD $= 0$
$3d+6$. HALT

Similarly for OR and NOT gates. Input and constant gates are even easier, implemented by a single READ or LOAD = instruction. It is easy to prove now by induction on d that, after executing these instructions, Register i (Processor i's accumulator) will contain the correct value of gate g_i. We make sure that the output gate is always g_1 so that the final answer is left on Register 1.

To achieve a better processor count we employ again Brent's principle: We first compute the number $q(n) = \lceil \frac{g(n)}{f(n)} \rceil$. For each value of d we make a list of the gates for which d is the length of the longest path from any input gate. We then assign these gates to the $q(n)$ processors as equitably as possible. The

values of the gates are will now have to be kept not in the accumulators, but in separate registers. □

It is a little more surprising that circuits can simulate PRAMs quite efficiently:

Theorem 15.2: Suppose that a function F can be computed by a uniform PRAM in parallel time $f(n)$ with $g(n)$ processors, where $f(n)$ and $g(n)$ can be computed from 1^n in logarithmic space. Then there is a uniform family of circuits of depth $\mathcal{O}(f(n)(f(n) + \log n))$ and size $\mathcal{O}(g(n)f(n)(n^k f(n) + g(n)))$ which computes the binary representation of F, where n^k is the time bound of the logarithmic-space Turing machine which on input 1^n outputs the nth PRAM in the family.

Proof: For fixed size n of the binary representation there are at most $g(n)$ processors in the corresponding PRAM. As we have argued in the proof of Theorem 2.5 for RAM's (see the Claim there), the PRAM's registers contain integers of length bounded by $\ell(n) = n + f(n) + b$, where b is the length of the longest integer explicitly referred to in the PRAM program—certainly at most n^k, because all such integers must be generated in logarithmic space on input 1^n. Also, the number of instructions in each RAM program is also bounded by a polynomial in n. Since we have at most $g(n)$ RAMs working for at most $f(n)$ steps, we know that at most $f(n)g(n)$ registers will be affected during the computation.

As a result of all this, the configuration $C = (\kappa_1, \kappa_2, \ldots, \kappa_{q(m,n)}, R)$ of the PRAM can be encoded in $\mathcal{O}(g(n)f(n)\log n)$ bits. R encodes the contents of the memory, given as pairs of the form (location, contents). All integers are in binary, each with enough leading zeros to attain the maximum of $\ell(n)$ bits. Thus, C is a sequence of bits, where it is *a priori* known which bits correspond to the ith program counter, and which bits encode the rth location-contents paitr in R.

The question is, how can we compute the encoding of the next configuration from that of the current configuration? We shall argue that this can be done very fast in parallel—and thus by a circuit of small depth. Each RAM instruction is easy to implement. Suppose for example that we know that the current instruction of RAM i is "t: ADD j," and we know the precise bits in the encoding of the configuration where the contents of Registers i and j are encoded. Then we have to compute the sum of these two integers in $\log \ell$ parallel time and $\mathcal{O}(\ell)$ work, and replace the contents of i by the sum. There are three problems: First, we do not know which instruction is executed in RAM i; we have to look this up from the program counter κ_i contained somewhere in the configuration. Second, we do not know where in the encoding of the configuration to look for the contents of j; we must examine all pairs in R. Third, other RAMs may be competing to write in Register i, and the lowest-indexed

one must prevail.

The first two problems can be solved by redundancy: To continue the example of the "t: ADD j" instruction, we have the following algorithm, for each $r \leq f(n)g(n)$:

"If program counter κ_i is t, and if the rth pair in the encoding R of the register contents is of the form (j, x) then Register i is incremented by x."

The two tests of this algorithm can be implemented easily by circuits of depth $\log \ell$ and size $\mathcal{O}(\ell)$ that take the corresponding bits of the current configuration as input, and compute a Boolean output, basically performing bitwise comparisons. If both of these outputs are **true**, then the addition is implemented. Notice that we must have such a circuit for each instruction in each RAM program, and for each (location, contents) pair—a total of $\mathcal{O}(n^k f(n)g(n))$ circuits, where the n^k part corresponds to the total number of instructions in the PRAM, bounded by the running time of the Turing machine that constructs it.

The third problem that we identified (the write-write conflicts) can be solved by first recording all writes done in the current step (at most $g(n)$ of them) in a (location, writer, contents) format, and then resolving any conflicts by $g(n)^2$ integer comparisons, each of $\log \ell(n)$ depth.

We can implement all other RAM instructions in a manner similar to that we explained for "ADD j". Indirect addressing instructions (such as "ADD $\uparrow j$" can be implemented in two stages (first find out the contents of Register j, and then the contents of that register). And READ instructions will have to look up the input of the circuit.

We conclude that there is a circuit of depth $\mathcal{O}(\log \ell) = \mathcal{O}(\log f(n) + \log n)$ and of size $\mathcal{O}(g(n)(n^k f(n) + g(n))(\log f(n) + \log n))$ which, given the encoding of a PRAM configuration, computes the encoding of the next one. Finally, the circuit C_n that simulates the given PRAM, specializing on inputs of length n, consists of $f(n)$ cascaded copies of this circuit (recall that $f(n)$ can be computed in logarithmic space). □

15.3 THE CLASS NC

Define now

$$\mathbf{NC} = \mathbf{PT/WK}(\log^k n, n^k)$$

to be the class of all problems solvable in polylogarithmic parallel time with polynomial amount of total work. Taking the union over all exponents k ensures, as with **P** and **NP**, that this class is stable and robust with respect to variations of the model. For example, it follows from Theorems 15.1 and 15.2 that **NC** is precisely the class of languages decided by PRAMS in polylogarith-

mic parallel time with polynomially many processors. We shall also see in the next subsection that **NC** is closed under reductions.

It has been argued that **NC** captures our intuitive notion of "problems satisfactorily solved by parallel computers," very much the same way that **P** has been claimed to capture the intuitive notion of efficient computability in the sequential context. However, the argument here is much less convincing, and certainly not as widely accepted as for **P**. One problem is this: In sequential computation the difference between polynomial and exponential algorithms is real and dramatic—simply because it is one exponential closer to us... For example, 2^n is much larger than n^3 for very accessible values, say $n = 20$. In contrast, although $\log^3 n$ is in theory asymptotically much smaller than $\sqrt{n}$, the difference starts to become felt only when $n = 10^{12}$. And for such values of n, the notion of "polynomial number of processors" is absurd.

Another problem with our definition of **NC** is that it is a class *of languages*. As it became clear in Section 15.1, in parallel computation the more interesting problems require substantial output. (And, as we shall see later in this section, in parallel computation the equivalence between such problems and their decision versions breaks down.) In fact, many authors define **NC** to be the class of *functions* computable in polylogarithmic parallel time and polynomial work, not languages decided. Instead, we shall be using the term "**NC** algorithm" to mean a parallel algorithm, perhaps with substantial output, that obeys these bounds.

More refined notions of parallel complexity are obtained by defining the following family of important subclasses of **NC**:

$$\mathbf{NC}_j = \mathbf{PT/WK}(\log^j n, n^k).$$

That is, $\mathbf{NC}_j$ is the subset of **NC** in which the parallel time is $\mathcal{O}(\log^j n)$; the free parameter k means that we allow any degree in the polynomial accounting for the total work. For example, we established in Section 15.1 that REACHABILITY is in $\mathbf{NC}_2$. In fact, $\mathbf{NC}_2$ is a perfectly good candidate for an alternative, more conservative, notion of "efficient parallel computation." Notice that the $\mathbf{NC}_j$s comprise a potential *hierarchy* of complexity classes. Recall that, in the sequential domain, classes such as $\mathbf{TIME}(n^j)$ do form a hierarchy of proper inclusions (Problem 7.4.8). But the corresponding statement for the **NC** hierarchy is, once more, an important unproven conjecture.

Since the amount of work involved in solving any problem in **NC** is by definition bounded by a polynomial, it is clear that $\mathbf{NC} \subseteq \mathbf{P}$. But is $\mathbf{NC} = \mathbf{P}$? This important open question is the counterpart, for parallel computation, of the $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ conundrum. In both instances we are asking whether the class of satisfactorily solved problems (**P** in the sequential domain, **NC** in the parallel) is indeed a proper subset of a larger class, which is the natural limit of our

ambition (**NP** then, **P** now). As with the $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ problem, intuition and experience seems to suggest a negative answer: It would be truly remarkable if all polynomial-time solvable problems could be massively parallelized. Persistent failures to develop **NC** algorithms for some fairly simple problems in **P** (the maximum flow problem of the previous section, to name one) seem to imply that *there are problems that are inherently sequential*, indeed that $\mathbf{NC} \neq \mathbf{P}$. Unfortunately, no such proof is in sight. Thus, in order to identify probable "inherently sequential problems" we must once more turn to reductions and complete problems.

P-completeness

Among all problems in **P**, the **P**-complete problems are the least likely to be in **NC**—the most likely to be "inherently sequential." But to argue this is the case, we must first prove that our logarithmic-space reductions preserve parallel complexity. This is an instance of a more general principle, the *parallel computation thesis*, that relates space and parallel time (see Theorem 16.1):

Theorem 15.3: If L reduces to $L' \in \mathbf{NC}$, then $L \in \mathbf{NC}$.

Proof: Let R be the logarithmic-space reduction from L to L'. It is not hard to see that there is a logarithmic space-bounded Turing machine R' which accepts input (x, i) (where i is the binary representation of an integer no larger than $|R(x)|$) if and only if the ith bit of $R(x)$ is one. Now by solving the reachability problem for the configuration graph of R' on input (x, i) we can compute the ith bit of $R(x)$. Therefore, if we solve all these problems in parallel by $\mathbf{NC}_2$ circuits, we can compute all bits of $R(x)$. Once we have $R(x)$ we can use the **NC** circuit for L' to tell whether $x \in L$, all in **NC**. □

Notice that our proof implies the following refinement:

Corollary: If L reduces to $L' \in \mathbf{NC}_j$, where $j \geq 2$, then $L \in \mathbf{NC}_j$. □

As we have already observed in Section 15.1, computing the maximum flow in a network is a task that seems to be inherently sequential. We shall prove that the following problem is **P**-complete:

ODD MAX FLOW: Given a network $N = (V, E, s, t, c)$, is the maximum flow value odd?

Obviously, if we cannot decide this problem in **NC**, then neither can we compute the maximum flow value by an **NC** algorithm. Notice that we are using a non-standard decision problem here as a surrogate of the optimization problem, instead of our familiar MAX FLOW (D), asking whether the maximum flow value is greater than a given goal—MAX FLOW (D) can also be shown **P**-complete via a somewhat more complicated reduction, see Problem 15.5.4. As we shall see in the next subsection, in parallel computation the equivalence between optimization problems and their decision versions breaks down in a most interesting

way—and so we should have little regret for abandoning MAX FLOW (D) in this case.

Theorem 15.4: ODD MAX FLOW is **P**-complete.

Proof: We know that it is in **P** (recall Section 1.2). To show completeness, we shall reduce MONOTONE CIRCUIT VALUE to the ODD MAX FLOW problem.

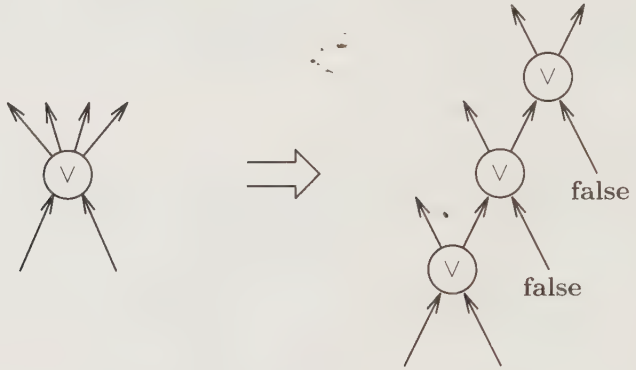
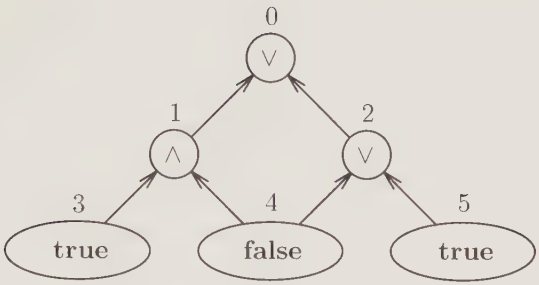


Figure 15-1. Reducing the fan-out of gates.

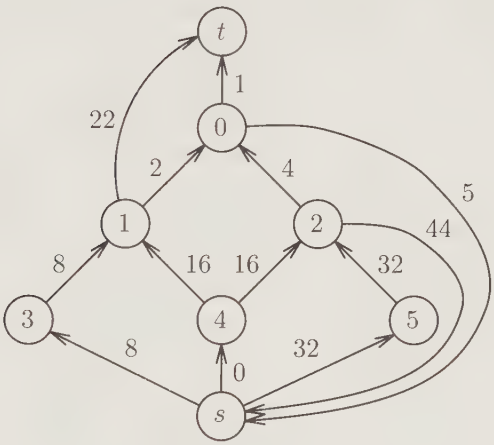
We are given a monotone circuit C. Assume that the output gate of C is an OR gate, and that no gate of C has outdegree more than two. This latter requirement can be assured by introducing at each gate with outdegree $k > 2$ a set of $k - 2$ OR gates, arranged in a tree, so that the other input of each such gate is a new **false** gate (see Figure 15.1). Assume further that the gates of C have been given consecutive numbers $0, 1, \ldots, n$, so that each gate has a smaller label than its predecessor; thus the output gate will have label 0, and the larger labels will be assigned to the inputs (see Figure 15.2(a)).

Our construction is this: The network $N = (V, E, s, t, c)$ produced from C has as its set of nodes the gates $0, \ldots, n$, plus two new nodes s and t (the source and the sink). We shall next describe the edges leaving each node, and their capacities. First, from the source s there is an edge going to each **true** gate i, and the capacity of this edge is $d2^i$ — two raised to the label of the **true** gate, multiplied by the outdegree d of the gate. From a **true** or **false** gate there is an edge of capacity 2^i to each successor gate. From an OR or AND gate i there is an edge of capacity 2^i to each gate that has it as a predecessor. From the output gate there is an edge of capacity one to vertex t.

Consider any AND or OR gate i. It already has several incoming and outgoing edges. Notice that, since it has at most two outgoing edges of capacity 2^i, and the capacities of each of the two incoming edges is at least twice that (the labels of its predecessors are strictly larger than i), there is a *surplus* of



(a)



(b)

Figure 15-2. The construction of the network.

incoming capacity, denote it $S(i)$. If i is an AND gate, there is an edge (i, t) of capacity $S(i)$; if it is an OR gate, then there is an edge (i, s) of capacity $S(i)$. This completes the construction; see Figure 15.2(a) and (b) for an example. Notice that we have edges going into s, although we know that such edges are superfluous in the max-flow problem; they are handy in the proof below.

Fix a flow f. A gate is called *full* with respect to this flow if all of its outgoing edges to its successors gates are filled to capacity; it is called *empty* if all of these edges have zero flow. Flow f is called *standard* if all gates that have value **true** are full, and all gates that have value **false** are empty (see Figure 15.3; full gates are shown in bold).

We claim that *a standard flow always exists*, and that *it is in fact the maximum flow*. To construct a standard flow, first fill all edges out of s to capacity. Then process all gates starting from the inputs and proceeding to the output. All **true** input gates have enough flow to become full (recall that the capacity of the incoming edge is $d2^i$), and all **false** input gates obviously must be empty (no incoming flow). This starts our induction on the depth of the gates. All OR gates that have value **true** have, by induction, at least one incoming edge filled to capacity, and thus enough flow to fill their outgoing edges and perhaps part of the surplus edge going back to s. All OR gates that have value **false** have no incoming flow, because their predecessors are empty by induction, and thus must be empty. All AND gates with value **true** have both incoming edges filled, and thus have enough flow to fill all outgoing edges (including the surplus edge to t); and finally all AND gates that have value **false** have at most one incoming edge filled with flow, which they can direct to the surplus edge (it has enough capacity to handle it).

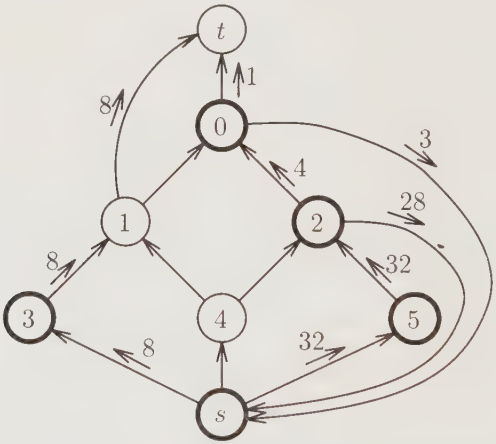


Figure 15-3. Standard flow.

We next claim that the standard flow f has maximum value. In proof, suppose that we separate the nodes of N into two groups: The first group contains s and the gates with value **true**, and the second group contains t and the gates with value **false**. *We claim that this cut has capacity equal to the value of* f, establishing the optimality of f. But this is easy to check: There are two kinds of edges going from the first group to the second, either (a) edges from true OR gates (or true input gates) to false AND gates, or (b) from true AND gates, or the output gate, if it is true, to t. Both kinds of edges are full in f, and they together account for all flow coming into t. Hence the capacity

of this cut equals the value of f, and thus f is maximum (recall the max-flow min-cut theorem, Problem 1.4.11).

Finally, notice that all flow values in the standard flow are even integers *with the single possible exception of the edge from the output gate to t.* Hence, the value of the maximum flow is odd if and only if the output gate is full, which happens if and only if the value of the output is **true**. □

15.4 RNC ALGORITHMS

There is a large and growing list of **P**-complete problems (see the references). As it is considered very unlikely that any of these problems can be solved efficiently in parallel, research on parallel algorithms for these problems has been redirected to less ambitious goals, very much the same way as with **NP**-complete problems: Parallel approximation algorithms, parallel algorithms for special cases, parallel heuristics.

There is an important problem that we know is a special case of MAX FLOW: The problem of telling whether a graph has a perfect matching (recall the construction in Figure 1.6 of Section 1.2). Unfortunately, we do not know whether MATCHING is in **NC**. However, we have accumulated all the ingredients necessary for a *randomized* parallel algorithm for matching: Just run the Monte Carlo algorithm for matching based on symbolic determinants (Section 11.1), using the **NC** algorithm for computing numerical determinants (Section 15.1). We conclude that telling whether a bipartite graph has a perfect matching belongs to **RNC**, the randomized version of **NC** (compare with **P** and **RP**), defined next.

Definition 15.3: A language L is in **RNC** if there is a uniform family of **NC** circuits, with the following additional properties: First, the circuit C_n specializing in strings of length n has now $n + m(n)$ gates, where $m(n)$ is a polynomial—intuitively, the additional gates are the random bits needed for the algorithm. If a string x of length n is in L, then at least half of the $2^{m(n)}$ bit strings y of length $m(n)$ the circuit C_n outputs **true** on input $x; y$. If $x \notin L$, C_n outputs **false** on $x; y$ for all y. □

To resume our discussion of MATCHING, we can tell efficiently in parallel whether a bipartite graph has a perfect matching. But this is not yet satisfactory, because what we really need is to *compute* the perfect matching, not just to be assured of its existence. The only reason why we are studying decision problems is because they are usually equivalent to the corresponding search problems. That is, given an efficient algorithm that decides whether a solution exists, there is a general "dynamic programming technique" (recall Example 10.3 for 3SAT and 10.4 for the TSP) that actually computes the desired solution, if of course it exists. The catch is that *this method is inherently sequential.* There is no general efficient parallel method for turning decision algorithms

into algorithms that actually find the desired solution.

Fortunately, in the case of matching a clever trick works. It is best explained in terms of a more general problem, the *minimum-weight perfect matching* problem. Suppose that each edge $(u_i, v_j) \in E$ has a weight w_{ij} associated with it, and we are seeking not just any perfect matching, but the perfect matching π that minimizes $w(\pi) = \sum_{i=1}^{n} w_{i,\pi(i)}$. It turns out that there is an **NC** algorithm for this problem, which works under two conditions: First, the weights must be small, polynomial in n. Second, *the minimum-weight matching must be unique.*

This algorithm further exploits the connection between matchings and determinants. We define a matrix $A^{G,w}$ whose i,jth element is $2^{w_{ij}}$ if (u_i, v_j) is an edge, and 0 otherwise. That is, we replace the entries x_{ij} of A^G with two raised to the power of the weight of the edge (this is why we need the weights to be polynomial). What is the determinant of $A^{G,w}$? Recall that

$$\det A^{G,w} = \sum_{\pi} \sigma(\pi) \prod_{i=1}^{n} A^{G,w}_{i,\pi(i)}.$$

First notice that all terms associated with permutations that are not perfect matchings of G are zero. Thus the summation ranges over all perfect matchings. Second, the term $\prod_{i=1}^{n} A^{G,w}_{i,\pi(i)}$ is precisely $2^{w(\pi)}$. In other words, $\det A^{G,w}$ is the sum of powers of two (perhaps negated because of the $\sigma(\pi)$ factor), where the exponents are the weights of the perfect matchings.

Recall now that the minimum-weight perfect matching is unique; suppose its weight is w^*. Thus, all terms of $\det A^{G,w}$ are multiples of 2^{w^*}. And all of them but one will be *even multiples* of 2^{w^*}. In other words, $\det A^{G,w} = 2^{w^*}(1+2k)$ for some integer k (possibly negative). Thus 2^{w^*} is the *highest power of two that divides* $\det A^{G,w}$. Based on this fact, we can calculate w^* efficiently in parallel as follows: We first calculate $\det A^{G,w}$ using our **NC** algorithm for determinants (this number has a polynomial number of bits, because the weights are polynomial); w^* is the number of trailing zeros† in the binary representation of $\det A^{G,w}$.

Once we have w^* we can test whether an edge (u_i, v_j) is in the minimum-weight perfect matching as follows: We delete this edge and its nodes from G, and compute the weight of the minimum matching in the resulting graph. It should be clear that edge (u_i, v_j) is in the minimum-weight matching of G if and only if the new minimum weight is precisely $w^* - w_{ij}$. We test all edges in parallel.

What we have proved so far is that *if the minimum-weight perfect matching exists and is unique, then it can be computed efficiently in parallel.* And we

† Strictly speaking, we must count in parallel the number of zeros in the end of the binary representation of the integer $\det A^{G,w}$; but this can be done easily by prefix sums.

know how to test in **RNC** whether a perfect matching exists. But how can one guarantee that the minimum-weight matching is unique? Here is where randomization helps: It turns out that, for randomly chosen small weights, the minimum-weight matching is unique with high probability:

Lemma 15.1 (The Isolating Lemma): Suppose that the edges in E are assigned independently and randomly weights between 1 and $2|E|$. If a perfect matching exists, then with probability at least $\frac{1}{2}$ the minimum-weight perfect matching is unique.

Proof: Call an edge *bad* if it is in one minimum-weight matching but not in another. Obviously, the minimum-weight perfect matching is unique if and only if there are no bad edges.

Consider now an edge $e = (u_i, v_j)$, and suppose that all weights have been assigned except for es. Let $w^*[\overline{e}]$ be the smallest weight among all perfect matchings that do not contain e, and let $w^*[e]$ be the smallest weight among all perfect matchings that contain e, *but not counting the weight of* e. Consider the number $\Delta = w^*[\overline{e}] - w^*[e]$. Obviously Δ does not depend on the weight of e—it was defined this way.

We next draw the weight w_{ij} of e. *We claim that* e *is bad if and only if* $w_{ij} = \Delta$. The reason is simple: If $w_{ij} < \Delta$ then e is in every minimum-weight matching, and if $w_{ij} > \Delta$ then e is included in no minimum-weight matching; in both cases e is not bad. If $w_{ij} = \Delta$ the e is indeed bad, because both minimum matchings, the one that contains e and the one that doesn't, are now minima.

It follows that $\mathbf{prob}[e \text{ is bad}] \leq \frac{1}{2|E|}$, because this is the probability that a randomly drawn integer between 1 and $2|E|$, will coincide with Δ—the less-than-or-equal sign to remind us that Δ could be completely out of this range. Therefore the probability that there is *some* bad edge among the $|E|$ ones is at most $|E|$ times that bound, and thus no more than half.

Incidentally, notice that the proof of the lemma has little to do with matchings: It holds for any family of subsets, of which we wish to isolate one. □

Our algorithm for finding a perfect matching in a bipartite graph is now complete: Assign random weights to the edges, and run the algorithm that computes the minimum-weight matching if it is unique. If a perfect matching exists, with probability at least $\frac{1}{2}$ this algorithm will return one.

Small Capacities

We end this chapter with an interesting post scriptum on MAX FLOW: In order to prove that MAX FLOW is **P**-complete we had to use exponentially large capacities. On the other hand, we just saw that the matching problem, which can be considered as a special case of MAX FLOW with unit capacities (recall Figure 1.6), can be solved efficiently in parallel (with the help of randomization). This raises the question, whether there is an **RNC** algorithm for MAX FLOW *when*

the capacities are expressed in unary. We shall next show that indeed such an algorithm exists. In fact, it leads to an **RNC**-approximation scheme for MAX FLOW (see Problem 15.5.8; notice the striking analogy with KNAPSACK, recall Sections 9.4 and 13.1).

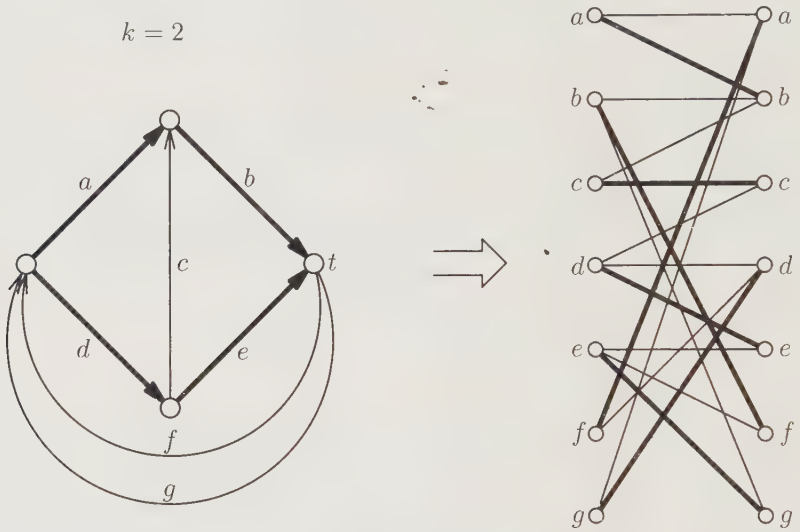


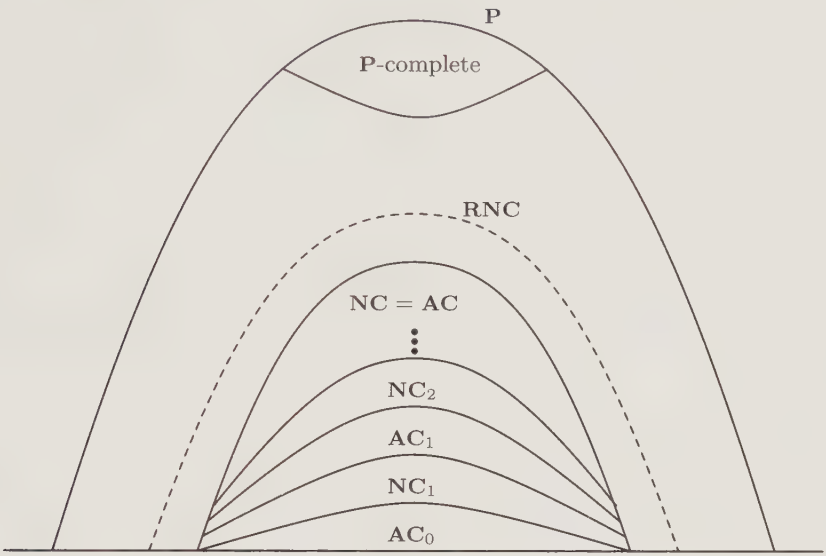
Figure 15-4. From max flow with unit capacities to matching.

Suppose that we are given a network with capacities in unary. Equivalently, we can assume that in the given network all edges have unit capacity (capacity greater than one can be achieved by allowing for a moment *multiple edges*). We add k parallel edges from the t to s, where k is the flow value to be achieved (Figure 15.4).

Create now a bipartite graph, in which both sets of nodes coincide with the set of edges of the network (including the k new ones, see Figure 15.4). There is an edge (e, e') in the bipartite graph if and only if the head of e is the tail of e'. Also, we add all edges of the form (e, e), where e is an edge of the original network (intuitively, this will allow any old edge to carry zero flow, but not a new one). It is not hard to check that *there is a perfect matching in the resulting bipartite graph if and only if there is a flow of value k in the original network* (see Figure 15.4 for an example.) Naturally, now that we know how to solve the decision problem in **RNC**, binary search will yield the optimum solution.

15.5 NOTES, REFERENCES, AND PROBLEMS

15.5.1 Class review:



15.5.2 For much more on parallel algorithms, and parallelism in general, see

- S. Akl *The Design and Analysis of Parallel Algorithms,* Prentice Hall, Englewood Cliffs, 1989,
- J. Já Já *An Introduction to Parallel Algorithms,* Addison-Wesley, Reading, Massachusetts, 1992, and
- R. M. Karp and V. Ramachandran "Parallel algorithms for shared-memory machines," pp. 870–941 in *The Handbook of Theoretical Computer Science, vol. I: Algorithms and Complexity*, edited by J. van Leeuwen, MIT Press, Cambridge, Massachusetts, 1990,

among several other books and review articles. The determinant algorithm explained in Section 15.1 is from

- A. L. Chistov "Fast parallel evaluation of the rank of matrices over a field of arbitrary characteristic," *Fund. of Computation Theory,* Lecture Notes in Computer Science, Volume 199, Springer Verlag, Berlin, pp. 63–79, 1985.

For a formal and thorough treatment of parallel algorithms in terms of models of parallel computation that are closer to today's massively parallel computers see

- F. T. Leighton *Parallel Algorithms and Architectures,* Morgan-Kaufman, San Mateo, California, 1991.

15.5.3 Problem: Show that if $\mathbf{NC}_{i+1} = \mathbf{NC}_i$, then $\mathbf{NC} = \mathbf{NC}_i$. That is, if two consecutive levels of the **NC** hierarchy coincide, the whole hierarchy collapses to that

level. (Compare with Theorem 17.9; in fact, the proofs of the two results are not unrelated.)

15.5.4 Recall that a Boolean *expression* can be considered as a circuit in which each gate is used as an input to at most one other gate. Obviously, circuits can be more succinct and economical than expressions, but the question is *by how much?* For example, can a circuit be converted into an equivalent expression with only polynomial increase in its size? It turns out that this question is a restatement of the fundamental question about parallel computation!

Problem: Show that a language can be computed by a uniform family of expressions with polynomial size if and only if it is in $\mathbf{NC}_1$. (This was first pointed out in

- P. M. Spira "On time-hardware complexity tradeoffs for Boolean functions," *Proc. 4th Hawaii Conference on Systems Sciences,* pp. 525–527, 1971.

Starting from any polynomial-sized expression, even one with polynomial depth, one can "rebalance" it into an equivalent expression with logarithmic depth; see also

- R. P. Brent "The parallel evaluation of general arithmetic expressions," *J.ACM 21*, pp. 201–206, 1974

for a more general application of this important technique.) Therefore, unless $\mathbf{P} = \mathbf{NC}_1$, circuits cannot be simulated by expressions with only a polynomial loss of size.

For a direct proof that any polynomial-size expression can be evaluated in a variant of $\mathbf{NC}_1$ see

- S. R. Buss "The Boolean formula value problem is in ALOGTIME," *Proc. 19th ACM Symp. on the Theory of Computing*, pp. 123–131, 1987.

For another surprising characterization of $\mathbf{NC}_1$ see

- D. Barrington "Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1," *Proc. 18th ACM Symp. on the Theory of Computing*, pp. 1–5, 1986.

15.5.5 We have been careful to have circuits whose gates have bounded *fan-in* (that is, they each take at most two inputs), but we allow unbounded *fan-out* (each gate may feed in an arbitrary number of other gates.

Problem: Show that any circuit can be transformed into another with fan-out two, with only linear increase in size and depth. (This is from

- H. J. Hoover, M. M. Klawe, and N. Pippenger "Bounding fan-out in logical networks," *J.ACM 31*, pp. 13–18, 1984.)

15.5.6 The AC hierarchy. To go in the opposite direction of the previous problem, it makes sense to define circuits with AND and OR gates that have *unbounded fan-in.* That is, OR gates can have many inputs, and compute in one step the OR of all their inputs; similarly for AND. For example, although computing the OR of n bits requires circuits with logarithmic depth in the ordinary model, it can be achieved by depth one in the unbounded fan-in model. Now, for $i \geq 0$ define $\mathbf{AC}_i$ to be the class of languages that can be decided by uniform families of circuits with unbounded fan-in,

polynomial size, and depth $\mathcal{O}(\log^i n)$ (thus constant depth if $i = 0$). Define **AC** to be the union of all $\mathbf{AC}_i$s.

(a) Show that $\mathbf{NC}_i \subseteq \mathbf{AC}_i \subseteq \mathbf{NC}_{i+1}$. Conclude that $\mathbf{AC} = \mathbf{NC}$.

(b) Show that if $\mathbf{AC}_{i+1} = \mathbf{AC}_i$, then $\mathbf{AC}_j = \mathbf{AC}_i$ for all $j > i$ (recall Problem 15.5.3).

(c) Show that $\mathbf{AC}_i$ is precisely the class of languages that can be decided by uniform PRAM programs with polynomial processors and time $\mathcal{O}(\log^i n)$. (This is from

- L. J. Stockmeyer and U. Vishkin "Simulation of parallel random access machines by circuits," *SIAM J. Computing, 13*, pp. 409–423, 1984.)

Unlike $\mathbf{NC}_i$, the $\mathbf{AC}_i$ hierarchy has an interesting class already at its zeroth level: $\mathbf{AC}_0$ contains all languages that can be decided in *constant* depth and polynomial size by unbounded fan-in circuits. (Without the size bound, *all languages in* **P** could be thus decided, recall Theorem 4.1 on conjunctive normal form.) For an interesting proof that the *parity* language (all bitstrings with an odd number of ones) is not in $\mathbf{AC}_0$, see

- M. Furst, J. Saxe and M. Sipser "Parity, circuits, and the polynomial hierarchy," *Math. Systems Theory, 17*, pp. 13–27, 1984.

Interestingly, this same language seems to be difficult to solve by neural networks, see

- J. Hertz, A. Krog, and R. G. Palmer *Introduction to the Theory of Neural Computation*, Addison-Wesley, Reading, Massachusetts, 1991.

15.5.7 PRAM models. There are many variants of PRAM models, depending on what kind of simultaneous access to the memory is allowed. The PRAM that we described is the CRCW PRAM (for "concurrent read, concurrent write"), since multiple processors are allowed to read the same memory location simultaneously, and even to write it simultaneously. A weaker model would be CREW, for "concurrent read, exclusive write," where concurrent read access is still allowed, but only one processor can write at a location in every step. Finally, EREW allows no concurrent acces, read or write.

These three models represent three increasingly realistic assumptions about implementability of simultaneous memory access by hardware. In fact, there are three kinds of CRCW PRAM's, depending on how the value written in a memory location is selected among the many simultaneous writing attempts by the processors. We have discussed the PRIORITY CRCW PRAM, in which the winner is selected according to processor number. A weaker and more realistic mechanism would be the ARBITRARY CRCW PRAM model, in which the machine selects arbitrarily one of the values written (and the program must therefore be prepared to function correctly in the face of all outcomes). Finally, in the COMMON CRCW PRAM model all processors attempting to write the same location the same time must do so with the same, common value. We have therefore these *five* PRAM models, in order of increasing strength:

EREW CREW COMMON CRCW ARBITRARY CRCW PRIORITY CRCW

It turns out that the last three models are equivalent, if one disregards polynomial differences in the number of processors:

Problem: Show that a PRIORITY CRCW PRAM with time t and p processors can be simulated by a COMMON CRCW PRAM with time $\mathcal{O}(t)$ and $\mathcal{O}(p^2)$ processors.

There are results that show that each model among the first three is under some circumstances more powerful than the previous ones; see the next problems, and Section 3 of the survey paper by Karp and Ramachandran cited above. However, the performances of any two of these five machine models cannot differ by more than a logarithmic factor:

Problem: Show that a PRIORITY CRCW PRAM with time t and p processors can be simulated by an EREW PRAM with time $\mathcal{O}(t \log p)$ and $\mathcal{O}(p)$ processors.

15.5.8 A CROW PRAM (PRAM with concurrent read but with *own* write) is a PRAM allowing concurrent reads, in which each register is owned by a processor. The owner is the only processor that can write on it. Where in the spectrum above would you place CROW PRAMs? (CROW PRAMs, arguably the model closest to hardware realities, were proposed and studied in

- P. W. Dymond and W. L. Ruzzo "Parallel RAMs with owned memory and deterministic context-free language recognition," *Proc. 13th Intern. Conf. on Automata, Languages, and Programming*, pp. 95-104, 1986.

There is a surprising alternative characterization of their power in terms of deterministic context-free languages, see the paper above.)

15.5.9 Problem: (a) Suppose that we wish to compute the OR of n bits by a PRAM. Argue that a CRCW PRAM can do this in $\mathcal{O}(1)$ time with $\mathcal{O}(n)$ processors, while a CREW PRAM can do it in $\mathcal{O}(\log n)$ time with $\mathcal{O}(n)$ processors.

(b) Prove that $\Omega(\log n)$ steps are required of a CREW PRAM to compute the OR of n bits—even if each processor can perform arbitrary functions of its own register in unit time. (This is from

- S. A. Cook, C. Dwork, and R. Reischuk "Upper and lower bounds for parallel random access machines without simultaneous writes," *SIAM J. Comp., 15*, pp. 87–97, 1986.

Define what it means for an input bit to *affect* a processor of register at time t. Prove by induction on t that the number of input bits affecting any processor or register at time t cannot be more than c^t. Interestingly, c *has* to be larger than two.)

15.5.10 There are many more formal models of parallelism in the literature. During the middle 1970's a sequence of powerful models of computation exhibiting aspects of parallelism were proposed independently by several researchers. These included extensions of Turing machines

- W. J. Savitch "Recursive Turing machines" *Intern. J. Comp. Math., 6*, pp. 3–31, 1977,

random access machines with "vector processing" capabilities

- V. R. Pratt and L. J. Stockmeyer "Characterization of the power of vector machines," *J.CSS, 12*, pp. 198–221, 1976; see also

- J. Trahan, V. Ramachandran, and M. C. Loui "The power of random access machines with augmented instruction sets," in *Proc. 4th Annual Conf. on Structure in Complexity Theory*, pp. 97–103, 1989,

alternating machines (see the references in the next chapter), and several others. All these models shared the curious characteristic that for them polynomial time coincides with nondeterministic polynomial time! Naturally, the reason was for each of these models that both classes are the same as our **PSPACE**, a manifestation of the *parallel computation thesis*—see the next chapter. For formal treatments of parallelism that are closer in spirit to the massively parallel computers that are now commercially available (in that they treat the communication delay between processors in a less cavalier way than PRAM's) see

- P. W. Dymond and S. A. Cook "Hardware complexity and parallel complexity," *Proc. 21st IEEE Symp. on the Foundations of Computer Science*, pp. 360–372, 1980,
- L. G. Valiant "General purpose parallel architectures," pp. 953–971 in *The Handbook of Theoretical Computer Science, vol. I: Algorithms and Complexity*, edited by J. van Leeuwen, MIT Press, Cambridge, Massachusetts, 1990,
- C. H. Papadimitriou and M. Yannakakis "Towards an architecture-independent analysis of parallel algorithms," *Proc. 20th ACM Symp. on the Theory of Computing*, pp. 510–513, 1988,

and the book by Tom Leighton referenced above. See also

- P. van Emde Boas "Machine models and simulations," pp. 1–61 in *The Handbook of Theoretical Computer Science, vol. I: Algorithms and Complexity*, edited by J. van Leeuwen, MIT Press, Cambridge, Massachusetts, 1990

for a comprehensive comparative survey of computational models, including parallel ones.

15.5.11 For discussions of the class **NC** as a notion of feasible parallel computation see

- N. Pippenger "On simultaneous resource bounds," *Proc. 20th IEEE Symp. on the Foundations of Computer Science*, pp. 307–311, 1979,
- S. A. Cook "Towards a complexity theory of synchronous parallel computation," *Enseign. Math., 27*, pp. 99–124, 1981, and
- S. A. Cook "A taxonomy of problems with fast parallel algorithms," *Inform. and Control, 64*, pp. 2–22, 1985.

15.5.12 Finding the maximum independent set in a graph is of course **NP**-complete. But suppose that we wish to find a *maximal* independent set. This can surely be done fast by the "greedy algorithm:" Repeatedly add any vertex to the set, and delete it and its neighbors from G, until G is empty. *Unfortunately, this algorithm is too sequential.* In fact, it was conjectured by Les Valiant that the maximal independent set problem is inherently sequential.

Consider however the following way of "telescoping" the greedy algorithm. At each stage do not add to the independent set being constructed a single node of the

current graph, *but a whole independent set* S. How do we choose S? Here is one idea from

- M. Luby "A simple parallel algorithm for the maximal independent set," *SIAM J. Comp., 15*, pp. 1036–1053, 1986:

First put in S each node in G with probability $\frac{1}{d}$, where d is the degree of the node. Then examine all edges with both endpoints in S. For each such edge, remove from S the node with smallest degree, breaking ties arbitrarily. Finally, add the remaining nodes of S to the maximal independent set being created, and delete them and their neighbors from G. Repeat until G is empty.

Problem: (a) Show that at each stage the expected number of deleted edges is at least $\frac{1}{16}$ of the edges of G.

(b) Conclude that the algorithm above is an **RNC** algorithm for the maximal independent set problem.

In fact, Mike Luby proves that randomness is not needed at all in this algorithm: All that is required is that the random insertions of nodes in S be *pairwise* independent, which is a lot easier to guarantee. The random experiments now sample a *polynomially large* population, and so they can be replaced by exhaustive counting. Hence, this can be turned into a *deterministic* $\mathbf{NC}_2$ algorithm. Refinements of this technique of removing randomization in the context of parallelism, somewhat in the spirit of Problem 11.5.10, can be found in

- M. Luby "Removing randomness from parallel computation without processor penalty," *Proc. 29th IEEE Symp. on the Foundations of Computer Science*, pp. 162–173, 1988,
- B. Berger, J. Rompel "Simulating $\log^c n$-wise independence in NC," *Proc. 30th IEEE Symp. on the Foundations of Computer Science*, pp. 2–7, 1989, and
- R. Motwani, J. Naor, and M. Naor "The probabilistic method yields deterministic parallel algorithms," *Proc. 30th IEEE Symp. on the Foundations of Computer Science*, pp. 8–13, 1989.

However, the greedy algorithm does have an advantage over Luby's algorithm: It produces the *lexicographically first maximal independent set*, provided that the least node is inserted at each step. Apparently, this cannot be achieved fast in parallel:

(c) Show that finding the lexicographically first maximal independent set of a graph is **P**-complete. (This is a reduction from MONOTONE CIRCUIT VALUE; each gate is simulated by an edge; and the remaining edges, as well as the relative order of the nodes, reflect the structure of the circuit.)

15.5.13 Comparator circuits. (a) Show that the CIRCUIT VALUE problem for circuits with NAND gates is **P**-complete.

(b) Show that the CIRCUIT VALUE problem for circuits with only AND gates is in $\mathbf{NC}_2$. Repeat for $\oplus$ (exclusive-or gates).

A comparator gate has two inputs, x_1 and x_2, and two outputs: $(x_1 \vee x_2)$ and $x_1 \wedge x_2$.

(c) Design a circuit with four inputs, four outputs, and only comparator gates, which *sorts* its inputs.

Although all other conceivable circuit-value problems (see for example (a) and (b) above) can be classified as either **P**-complete or in **NC**, the circuit-value problem for circuits with comparator gates *seems to be somewhere in between.* For example, it can be evaluated in parallel time $\sqrt{n}$, while no **P**-complete problem is known to be thus solvable. See

- E. W. Mayr and A. Subramanian "The complexity of circuit value and network stability," *Proc. 4th Annual Conf. on Structure in Complexity Theory*, pp. 114–123, 1989.

(d) Show that the problem of finding the *lexicographically first maximal matching* in a graph (equivalently, the lexicographically first independent set in a *line graph*, see the previous problem and Problem 9.5.17) is equivalent to the circuit value problem for comparators.

15.5.14 Theorem 15.4 is from

- L. M. Goldschlager, R. A. Shaw, and J. Staples "The maximum flow problem is log space complete for P," *Theor. Comp. Science 21*, pp. 1073–1086, 1982.

Problem: Prove that MAX FLOW (D) is **P**-complete. (This is from

- T. Lengauer and K. W. Wagner "The binary network flow problem is log space complete for P," *Theor. Comp. Science 75*, pp. 357–363, 1990.)

15.5.15 For early **P**-completeness results see

- S. A. Cook "An observation on time-storage trade-offs," *Proc. 5th ACM Symp. on the Theory of Computing*, pp. 29–33, 1973; also, *J.CSS, 9*, pp. 308–316, and
- N. D. Jones and W. T. Laaser "Complete problems for deterministic polynomial time," *Theor. Computer Science 3*, pp. 105–118, 1976.

Here are two of them:

(a) A *path system* is a set of triples $T \subseteq V^3$, a generalization of directed graphs. We say that node i is *reachable* if either $i = 1$, or there are two (recursively) reachable nodes j, j' such that $(j, j', i) \in T$. PATH is this problem: "Given a path system, is node n reachable?" Show that it is **P**-complete.

(b) Recall the definition of a *context-free grammar* from Problem 3.4.2. CONTEXT-FREE EMPTINESS is this problem: "Given a context-free grammar G, is the language produced empty?" Show that it is **P**-complete.

15.5.16 For much more on **P**-completeness, as well as on parallel computation and complexity in general, see

- R. Greenlaw, H. J. Hoover, and W. L. Ruzzo *A Compendium of Problems Complete for P*, Oxford Univ. Press, in press, 1993,

a book that is something of a Garey and Johnson for **P**-completeness (and includes an extensive list of **P**-complete problems). As a general rule, however, proving problems **P**-complete is less of an esoteric art than **NP**-completeness (recall Chapter 9); the

reductions tend to be rather ordinary gadget constructions, starting from canonical forms of the CIRCUIT VALUE problem.

Problem: Show that the CIRCUIT VALUE problem remains **P**-complete even if (1) all of its gates other than the inputs are OR's and AND's, each with fan-in and fan-out equal to two; (2) the gates are arranged in levels; the zeroth level being the input gates, and the remaining levels alternating between levels of OR gates and levels with AND gates.

15.5.17 Problem: Show that the following two problems are **P**-complete: "Given a graph, does it have an induced subgraph which has (a) minimum degree at least k; (b) vertex connectivity at least k?" By "induced graph" we mean a subset of the nodes, and all the edges with both endpoints in the subset; "vertex connectivity" of a graph is the minimum number of vertices whose deletion disconnects the graph. (Part (a) is from

- R. J. Anderson and E. W. Mayr "Parallelism and greedy algorithms," in *Advances in Computing Research, vol. 4*, pp. 17–38, 1987, while (b) is from
- L. M. Kirousis, M. J. Serna, and P. Spirakis "The parallel complexity of the subgraph connectivity problem," *Proc. 30th IEEE Symp. on the Foundations of Computer Science*, pp. 163–175, 1988.

15.5.18 The **RNC** algorithm we presented for matching is from

- K. Mulmuley, U. V. Vazirani, and V. V. Vazirani "Matching is as easy as matrix inversion," *Proc. 19th ACM Symp. on the Theory of Computing*, pp. 345–354, 1987; also, *Combinatorica 7*, pp. 105–113, 1987.

It turns out that matching is also in **coRNC**:

- H. Karloff "A Las Vegas algorithm for maximum matching," *Combinatorica 6*, pp. 387–392, 1986.

Problem: (a) Show that the weighted matching problem, with polynomially bounded weights, can be solved in **RNC**.

(b) Describe an **RNC** approximation scheme for MAX FLOW.

The parallel complexity of the weighted version of matching with unbounded, binary weights is very much open.

15.5.19 Communication complexity. Suppose that Alice and Bob from Chapter 12 wish to evaluate a Boolean function $f(X, Y)$, where $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_n\}$ are two disjoint sets of Boolean variables. They each have unrestricted computing power, and they are both honestly interested in computing the true value of $f(X, Y)$. The problem is that Alice only knows the values of the variables in X, Bob those in Y, *and communication between them is very costly.*

They engage in a *communication protocol*, as follows: Alice starts by computing an arbitrarily complex Boolean function $a_1(X)$ and sends the bit a_1 to Bob; Bob computes an arbitrary Boolean function $b_1(Y, a_1)$ and sends the bit b_1 back to Alice. And so on, with Alice computing $a_{i+1}(X, b_1, \ldots, b_i)$ and Bob $b_{i+1}(Y, a_1, \ldots, a_{i+1})$. After k such exchanges, hopefully a lot fewer than n, the two have enough information

to agree on the value of $f(X,Y)$. *The minimum such k is called the communication complexity of f.*

(a) What is your estimate of the communication complexity of the following functions? (1) $f(X,Y) = 1$ if and only if $X = Y$; (2) $f(X,Y)$ is the total number of ones in X and Y; (3) $f(X,Y)$ is the total number of ones in X and Y *modulo two.*

(b) Suppose now that, before getting the inputs and running the protocol, Alice and Bob get to choose *how to best partition the bits in $X \cup Y$* to minimize communication complexity. Repeat Part (a) in this regime.

(c) We can define *nondeterministic communication protocols*, in which Alice and Bob make nondeterministic choices. As with any nondeterministic computation of functions, some computations may be unsuccessful, but all successful computations produce the right answer, and at least one computation is successful. Repeat (a) for nondeterministic communication complexity.

Communication complexity was proposed by

- A. C.-C. Yao "Some complexity questions related to distributive computing," *Proc. 11th ACM Symp. on the Theory of Computing*, pp. 294–300, 1979;

it captures the difficulty of computing Boolean functions by integrated circuits, and lower bounds on communication complexity can be translated easily in that domain:

- T. Lengauer "VLSI Theory," pp. 837–868 in *The Handbook of Theoretical Computer Science, vol. I: Algorithms and Complexity*, edited by J. van Leeuwen, MIT Press, Cambridge, Massachusetts, 1990.

Surprisingly, it can be shown that deterministic and nondeterministic communication complexity are related *quadratically,* very much like space; see

- A. V. Aho, J. D. Ullman, and M. Yannakakis "On notions of information transfer in VLSI circuits," *Proc. 15th ACM Symp. on the Theory of Computing*, pp. 133 - 139, 1983.

However, when we minimize communication complexity over all partitions of the input (as in (b) above), an exponential gap can be proved, see

- C. H. Papadimitriou and M. Sipser "Communication complexity," *Proc. 14th ACM Symp. on the Theory of Computing*, pp. pp. 196 200, 1982; also, *J.CSS, 28*, pp. 260–269, 1984.

For a comprehensive treatment of communication complexity as a parallel to time complexity see

- B. Halstenberg and R. Reischuk "Relations between communication complexity classes," *J.CSS, 41*, pp. 402 429, 1990.

There is an unexpected relationship between communication complexity and parallel complexity: Suppose that we wish to compute the function F with n inputs and one output in parallel. Perform now the following experiment: Alice is given an input $X = \{x_1, \ldots, x_n\}$ such that $F(X) =$ **true**, and Bob an input $Y = \{y_1, \ldots, y_n\}$ such that $F(Y) =$ **false**. They must produce an index i such that $x_i \neq y_i$ (such an i must exist by our assumptions).

(d) Show that the communication complexity of this problem is $\Theta(d_F)$, where d_F is the depth of the shallowest Boolean circuit (or expression, since there is no bound on the size, the two are equivalent) that computes F. (Show how the layers of a Boolean circuit for F can simulate the stages of a protocol, and vice–versa.)

This, along with analogous results for monotone circuit depth and depth in unbounded fan-in circuits, was pointed out in

- M. M. Klawe, W. J. Paul, N. Pippenger, and M. Yannakakis "On monotone functions with restricted depth," *Proc. 16th ACM Symp. on the Theory of Computing*, pp. 480–487, 1984, and
- M. Karchmer and A. Wigderson "Monotone circuits for connectivity require superlogarithmic depth," *Proc. 20th ACM Symp. on the Theory of Computing*, pp. 539–550, 1988.

In the second paper this connection was used to prove a result analogous to Razborov's theorem (Theorem 14.6) for space-bounded computation, namely that REACHABILITY (obviously a monotone function of the adjacency matrix) cannot be solved by monotone circuits of depth less than $c \log^2 n$ for some $c > 0$.

CHAPTER

16 LOGARITHMIC SPACE

The question of the power of nondeterminism in space is much less dramatic than the same problem in the time domain, a distant echo of the P vs. NP problem. But historically it was the first such problem to be considered.

16.1 THE L$\stackrel{?}{=}$NL PROBLEM

As we have already seen, the innards of **P** are teeming with interesting complexity questions. The most classical of these concerns logarithmic space. Whether nondeterminism is more powerful than determinism in this context, that is, whether $\mathbf{L} = \mathbf{NL}$, is yet another important open question.

What we *do* know, however, is that both **L** and **NL** fall within **NC**. In fact, we understand almost precisely the intriguing intertwinement between logarithmic space classes and parallel complexity classes:

Theorem 16.1: $\mathbf{NC}_1 \subseteq \mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{NC}_2$.

Proof: The second inclusion is trivial. The third inclusion follows from the reachability method (recall Section 7.3): In order to determine whether input x is accepted by a nondeterministic logarithmic-space Turing machine N we simply produce the configuration graph of N on input x, and determine in $\mathbf{NC}_2$ whether an accepting node is reachable from the initial node (recall from the previous chapter that REACHABILITY is in $\mathbf{NC}_2$).

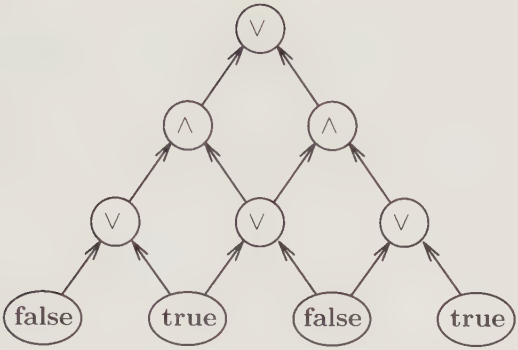
Now for the first inclusion. We must give an algorithm which evaluates in logarithmic space any uniform family of circuits with logarithmic depth. Our algorithm is the composition of three logarithmic-space algorithms (and we know from Proposition 8.2 how to compose logarithmic-space algorithms). The

first algorithm is the one that generates circuits in the given uniform family. We assume that the circuit is represented as a list of gates, where for each gate we are given its sort, as well as its list of *predecessors* (gates from which there is an edge to this gate). **true** and **false** gates have no predecessors, and NOT gates have just one. The two predecessors of an OR or AND gate are ordered, so that we can distinguish between the first and the second predecessor. The first gate in the list is the output gate.

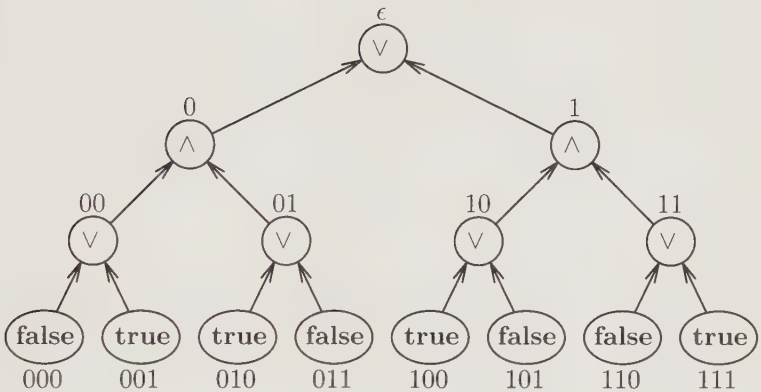
In the circuit a gate may have outdegree more than one (that is, it may be the predecessor of several gates; actually, this "sharing of common subexpressions" differentiates circuits from expressions, recall Section 4.3). The second logarithmic space-bounded algorithm takes this circuit and transforms it into an equivalent circuit with all outdegrees one (that is, essentially to an equivalent expression). This can be achieved as follows: We consider all possible *paths* in the original circuit, starting from the output and going towards the inputs. We do not represent a path by the names of the gates encountered (this would take $\log^2 n$ space), but by a bit string of length equal to that of the path, where each bit indicates whether the next gate visited in the path is the first or the second predecessor of the previous gate (the unique branch out of a NOT gate is denoted 0). Notice that, since the given circuit has logarithmic depth, these paths have logarithmic length.

Now, the equivalent tree-like circuit will have these paths as gates. That is, the output gate will be labeled ϵ, the empty string. Its first predecessor will be labeled 0, its second 1, the first predecessor of 1 will be labeled 10, and so on (see Figure 16.1). Gates reachable by several paths will be represented many times, once for every path that reaches them. The gates and the connections of the new circuit can be generated patiently, one-by-one, reusing space. In the end we have an equivalent tree-like circuit (Figure 16.1(b)) whose gates are labeled by bit strings of logarithmic length. That is, our new circuit representation is a list of bit strings, each with a sort.

Our third algorithm evaluates the output gate of the tree-like circuit. To evaluate an AND gate labeled by the string g, the algorithm recursively evaluates its first predecessor $g0$. If the outcome is **false**, then we need not evaluate the second input: We already know that the gate's value is **false**. But if the first predecessor's value is **true**, then we must also evaluate the second predecessor, $g1$. For OR gates the roles of **true** and **false** are reversed. For NOT gates we simply evaluate the unique input and return the opposite value, and in the case of **true** or **false** gates there is nothing to do. Once the evaluation of a gate is finished, the evaluation of its successor (the unique gate to which it is a predecessor, recall that we are evaluating a tree-like circuit) is resumed. The label of the successor can be obtained by simply omitting the last bit of the current label. When we finish the evaluation of the output, we have the value of the circuit and we are done.



(a)



(b)

Figure 16-1. A circuit (a) and an equivalent tree-like circuit (b).

How much information do we need to maintain in order to carry out this evaluation? The observation about **false** first predecessors of AND gates guarantees that *all we need to remember is the label of the currently evaluated gate*, and its value: If we are done evaluating the second predecessor of a gate, *the very fact that we had been evaluating the second predecessor* tells us the value of the first. It follows that the third algorithm correctly evaluates the circuit in logarithmic space, and the proof is complete. □

Theorem 16.1 is one way of stating the remarkably close relationship between space and parallel time: They are polynomially related. This impor-

tant observation has been termed "the parallel computation thesis." Naturally, it can be generalized beyond logarithmic space: $\mathbf{PT/WK}(f(n), k^{f(n)}) \subseteq \mathbf{SPACE}(f(n)) \subseteq \mathbf{NSPACE}(f(n)) \subseteq \mathbf{PT/WK}(f(n)^2, k^{f(n)^2})$. However only at $f(n) = \log n$ is the work needed polynomial.

The strongest result that we have delimiting the power of nondeterminism with respect to logarithmic space is Savitch's theorem (corollary to Theorem 7.5), implying that $\mathbf{NL} \subseteq \mathbf{SPACE}(\log^2 n)$. The "reachability method" used to establish this result, as well as $\mathbf{NL} = \mathbf{coNL}$ (Theorem 7.6) is a testimony to the close affinity of the REACHABILITY problem with nondeterministic space. Here is the full story:

Theorem 16.2: REACHABILITY is **NL**-complete.

Proof: We have already argued that REACHABILITY can be solved in logarithmic nondeterministic space (Example 2.10):

We shall show how to reduce any language $L \in \mathbf{NL}$ to REACHABILITY. The construction has been implicit in the reachability method: Suppose that L is decided by the $\log n$ space-bounded Turing machine N. Given input x, we can construct in logarithmic space the configuration graph of N on input x, denoted $G(N, x)$ (recall Section 7.3). We can assume that $G(N, x)$ has a single accepting node, call it n (with arcs from every accepting configuration to it); it certainly has a single initial node, call it 1. It is clear that $x \in L$ if and only if the produced instance of REACHABILITY has a "yes" answer. □

We have seen at least another interesting **NL**-complete problem:

Theorem 16.3: 2SAT is **NL**-complete.

Proof: We know (corollary to Theorem 9.1) that 2SAT is in **NL**. To prove completeness, we shall reduce UNREACHABILITY (the complement of REACHABILITY, and an **NL**-complete problem by virtue of $\mathbf{NL} = \mathbf{coNL}$) to 2SAT. First, we must start from a graph G that is *acyclic;* it is easy to see that the REACHABILITY problem is **NL**-complete even for such graphs (for example, see the proof of Theorem 16.5 below). We reduce the unreachability problem for such a graph to 2SAT by simulating each edge (x, y) of the graph by a clause $(\neg x \vee y)$, where we have a Boolean variable for each node in the graph. If we now add the clauses (s) and $(\neg t)$ for the start and target nodes s and t, it is clear that the resulting instance of 2SAT is satisfiable if and only if there is no path from s to t in the given graph. □

Does **L** have complete problems? The answer is positive, but completely uninteresting. Since a reduction is meaningful only within a class that is *computationally stronger than the reduction*, it seems that at **L** we have reached the limits of the usefulness of our logarithmic-space reductions: *All languages* in **L** are **L**-complete. To further categorize the languages of **L** we need weaker definitions of reductions (see Problem 16.4.4).

So, 2SAT is the satisfiability problem complete at this complexity level (complementing 3SAT for **NP**, HORN SAT for **P**, and more to come for other levels of complexity). Not unexpectedly, 2SAT also provides a precise logical characterization of **NL** in the spirit of Fagin's theorem for **NP** (recall Section 8.3): In analogy with Horn existential second-order logic (defined in Section 5.6), we call a sentence in existential second-order logic a *Krom sentence* (a "Krom clause" is an alternative term used in logic for a clause with two literals) if all of its first-order quantifiers are universal, and the matrix is a conjunction of clauses, each containing at most two atomic expressions that involve the second-order relation symbol. In the spirit of Theorems 8.3 and 8.4 we have:

Theorem 16.4: **NL** is precisely the class of all graph-theoretic properties expressible in Krom existential second-order logic with successor.

Proof: Problem 16.4.11. □

16.2 ALTERNATION

This is a good place to introduce an important generalization of nondeterminism, *alternation*. First, let us give an alternative definition nondeterminism in terms of configurations, as follows: A configuration "leads to acceptance" if and only if it is either a final accepting configuration, or (recursively) *at least one of its successors leads to acceptance*. That is, each configuration is in some sense an *implicit OR* of its successor configurations. In contrast, a machine deciding the complement of the same language would have configurations that are implicit ANDs.

Suppose now that we allow both modes in our nondeterministic machines. That is, some configurations are AND configurations and accept if all of their successors accept, while others are OR configurations, and accept if at least one of their successors does. The mode of each configuration (AND vs. OR) is determined by the state of the configuration. The machine accepts its input if and only if its initial configuration with this input does. We give the formal definition below:

Definition 16.1: An *alternating Turing machine* is a nondeterministic Turing machine $N = (K, \Sigma, \Delta, s)$ in which the set of states K is partitioned into two sets, $K = K_{\text{AND}} \cup K_{\text{OR}}$. Let x be an input, and consider the tree of computations of N on input x. Each node in this tree is a configuration of the precise machine, and includes the step number of the machine. Define now recursively, starting from the leaves of the tree and going up, a subset of these configurations, called *the eventually accepting configurations*, as follows: First, all leaf configurations with state "yes" are eventually accepting. A configuration C with state in K_{AND} is eventually accepting if and only if all of its successor configurations (configurations C' such that C yields in one step C') are eventually accepting. A configuration C with state in K_{OR} is eventually accepting if and only if at

least one of its successor configurations is eventually accepting. Finally, we say that N *accepts* x if the initial configuration is eventually accepting. We say that an alternating machine N *decides a language* L if N accepts all strings $x \in L$ and rejects all strings $x \notin L$.

We let $\mathbf{ATIME}(f(n))$ (*alternating time* $f(n)$) be the class of all languages decided by an alternating Turing machine, all computations of which on input x halt after at most $f(|x|)$ steps; $\mathbf{ASPACE}(f(n))$ (alternating space) is the class of all languages decided by an alternating Turing machine that uses no more than $f(|x|)$ space on input x. Finally, define $\mathbf{AP} = \mathbf{ATIME}(n^k)$ and $\mathbf{AL} = \mathbf{ASPACE}(\log n)$. □

The reader may be alarmed at the apparent proliferation of important complexity classes. Fortunately, we next prove a result that completely characterizes the power of alternating space in terms of much more familiar ideas: It turns out that alternating space *is precisely deterministic time, only one exponential higher.* (A very similar characterization of alternating *time* —it is roughly equivalent to deterministic space—will have to wait for three chapters.) One way to prove this important result makes use of complete problems, and in particular the MONOTONE CIRCUIT VALUE problem (Section 8.2):

Theorem 16.5: The MONOTONE CIRCUIT VALUE problem is **AL**-complete.

Proof: We first prove that the problem is in **AL**. The input of our alternating Turing machine is a circuit—say, given as a list of edges and sorts of the nodes. The machine examines the output gate of the circuit. If it is an AND gate, then the machine enters an AND state; if the output gate is an OR gate, then it enters an OR state. In either case, the machine determines the two gates that are predecessors of the output (it does so by remembering the output gate while examining all edges), and *it nondeterministically chooses one.* The same process is repeated at the new gate: The machine enters an AND or OR state depending on the sort of the gate, and looks for the gate's predecessors. And so on. If an input gate is encountered, the machine accepts if it is a **true** gate, and rejects if it is a **false** gate. Notice that in our description of the alternating machine, as in our programming of nondeterministic machines, only a few steps are true nondeterministic choices; we can think of the remaining steps as nondeterministic ones in which the choices are identical. The corresponding states can be AND states (OR states would work too, since both Boolean operations are idempotent).

Let us call the configurations that correspond to the machine examining a new gate the *gate configurations* of the computation. It follows by an easy induction on the height of a gate, using the recursive definition of eventual acceptance for alternating machines, that a gate configuration is an eventually accepting configuration if and only if the corresponding gate has value **true**. Hence the initial configuration is eventually accepting if and only if the output

is **true**, and the machine correctly evaluates the given circuit. Finally, it is clear that only logarithmic space is needed: The machine only has to remember the identity of the gate under consideration.

We must now show that any language $L \in \mathbf{AL}$ is reducible to the MONOTONE CIRCUIT VALUE problem. Consider such a language L, the corresponding alternating machine $M = (K_{\text{AND}}, K_{\text{OR}}, \Sigma, \Delta, s)$, and an input x. We shall construct a monotone circuit C such that the value of C's output is **true** if and only if N accepts x. Assume, as usual, that all transitions of N involve exactly two choices.

This construction is also straightforward (reflecting the close affinity of monotone circuits and alternating Turing machines). The gates of the circuit are all pairs of the form (C, i), where C is a configuration of N on input x, and i stands for the "step number," an integer between 0 and $|x|^k$, the time bound of the machine. The purpose of the step number is to make the circuit acyclic (configuration graphs in general may have cycles, whereas circuits don't). There is an arc from gate (C_1, i) to (C_2, j) if and only if C_2 yields in one step C_1 *and* $j = i + 1$. The sort of gate (C, i) depends on the state of the configuration C: If it is in K_{OR}, the gate is an OR gate; if it is in K_{AND} it is an AND gate; if it is "yes" then the gate is a **true** gate, and **false** if the state is "no". The output gate is the initial configuration on input x. It is clear, by the same correspondence as in the previous proof, that the circuit has output value **true** if and only if $x \in L$. □

Corollary 1: $\mathbf{AL} = \mathbf{P}$.

Proof: Both classes are closed under reductions, and they have the same complete problem (recall Proposition 8.4). □

In fact, the same argument can be used one exponential higher to show that polynomial alternating space is precisely **EXP** (Corollary 3 to Theorem 20.2), as well as even higher, or at any intermediate level:

Corollary 2: $\mathbf{ASPACE}(f(n)) = \mathbf{TIME}(k^{f(n)})$. □

16.3 UNDIRECTED REACHABILITY

The REACHABILITY problem for directed graphs is **NL**-complete, and thus it is not expected to be solvable in (deterministic) logarithmic space. But how about the same problem for *undirected graphs?* Since undirected graphs are a special kind of directed graphs, this problem may very well be easier. And it is: Although we do not know that UNDIRECTED REACHABILITY is in **L**, we shall establish that it can be solved in *randomized logarithmic space.*

Consider a language L, and a nondeterministic, logarithmic space-bounded Turing machine which decides L as follows: First, all of its computations halt on all inputs after the same number of steps (obviously a polynomial), and

there are two nondeterministic choices from every configuration—the machine is precise. More importantly, if $x \in L$, then at least half of its computations end up with "yes"; while, if $x \notin L$, all of the computations end up with "no". In other words, the machine is an **RP** machine which happens to use logarithmic space. **RL** is the class of all languages decided by such a machine.

Theorem 16.6: UNDIRECTED REACHABILITY is in **RL**.

Proof: Let $G = (V, E)$ be an undirected graph, and let $1, n \in V$. The randomized algorithm for telling whether there is an undirected path from 1 to n in V is very simple: It is *the random walk*. That is, we start at node 1, we choose an edge $[1, i]$ at random among all edges that leave 1, follow the edge to the new node i, and repeat[†]. For technical reasons that will become clear later, we assume that there is a chance that we may stay at the same node; that is, we assume that we have a *self-loop* edge $[i, i]$ at each node i.

Let v_t denote the node visited by the random walk at time t: $v_0 = 1$, and, if $v_t = i$ and $[i, j] \in E$, then $\mathbf{prob}[v_{t+1} = j] = \frac{1}{d_i}$, where by d_i we denote the *degree* of i, the number of edges (including the self-loop) incident upon it. Finally, let $p_t[i] = \mathbf{prob}[v_t = i]$ be the probability node i is visited at time t. Clearly, in the beginning of the random walk these probabilities largely depend on how close to 1 a node is. However, as time progresses they converge to something very simple:

Lemma 16.1: If $G = (V, E)$ is a connected graph, $\lim_{t\to\infty} p_t[i] = \frac{d_i}{2|E|}$ for all nodes i.

This is a remarkable result: It says that the probability of the random walk visiting a node at a particular time is proportional to the degree of the node (at least, this holds asymptotically, after many steps of the random walk have been taken). To put it otherwise, *all edges are equally likely to be traversed at any step, in either direction.*

Proof of the lemma: At time t the $p_t[i]$'s will deviate from the claimed asymptotic values $\frac{d_i}{2|E|}$; let $\delta_t[i] = p_t[i] - \frac{d_i}{2|E|}$ be this deviation at node i, and let $\Delta_t = \sum_{i\in V} |\delta_t[i]|$ be the total absolute deviation at time t.

How can we calculate the $p_{t+1}[i]$'s from the $p_t[i]$'s? Since the random walk is equally likely to visit each neighbor of the current node, we can think that the p_{t+1}'s are formed from the p_t's as follows: Each node i splits its $p_t[i]$ into d_i equal parts, where d_i is the degree of node i, and passes one such portion to each one of its neighbors (including itself, due to the self-loop). Each node i adds up the portions received from its neighbors, and the result is $p_{t+1}[i]$. But since $p_t[i] = \frac{d_i}{2|E|} + \delta_t[i]$, this splitting and passing can be thought of as *keeping*

[†] In Section 11.1 we showed that the random walk *on a path* solves 2SAT (Theorem 11.1). As we shall see, random walks on a regular graph take a little longer to converge, but not *too* long.

the $\frac{d_i}{2|E|}$ *part*, and splitting and passing only the $\delta_t[i]$'s splitting and passing the $\frac{d_i}{2|E|}$ part results in an amount equal to $\frac{1}{2|E|}$ being exchanged between any two neighbors, with canceling effects.

Since the $\delta_t[i]$'s are exchanged between adjacent nodes, the sum of the absolute values cannot increase. However, it can decrease *if two* $\delta_t[i]$*'s of opposite sign ever meet at a node*. We shall now show that they do:

Obviously, since at time t the total absolute deviation is Δ_t, there is a node i^+ with $\delta_t[j] \geq \frac{\Delta_t}{2|V|}$, and a node i^- with with negative deviation $\delta_k \leq -\frac{\Delta}{2|V|}$. There is a path $[i_0 = i^+, i_1, \ldots, i_m, \ldots, i_{2m} = i^-]$ with an even number of edges between i^+ and i^- (proof: if the shortest path between i^+ and i^- has an odd number of edges, add a self-loop to the path; self-loops are useful only at this point in the proof). The positive deviation from i^+ will travel along this path for m steps, always subdivided by the degree of the current node; similarly for the negative deviation. At least a positive deviation equal to $\frac{1}{|V|^m}$ of the original amount will arrive at the middle node i_m. Similarly for a negative deviation from the opposite direction. We conclude that after $m \leq n$ steps, a positive deviation of at least $\frac{\Delta}{2|V|^n}$ will cancel an equal amount of negative deviation, and thus in n steps the total absolute deviation has been decreased from Δ_t to at most $\Delta_t \cdot (1 - \frac{1}{|V|^n})$. Continuing like this, in the limit $\Delta_t \to 0$, and $p_t[i]$ converges to $\frac{d_i}{2|E|}$. □

This lemma, however, is an asymptotic result, and in fact one with exponentially slow convergence, and we only have polynomial time. Still, there is a way of using this result immediately: Stated another way, the lemma says that, asymptotically and on the average, *the walk returns to* i *every* $\frac{2|E|}{d_i}$ *steps*. Or, equivalently, if $v_t = i$, then *the expected time until the walk comes back to* i *for the first time after* t *is* $\frac{2|E|}{d_i}$. Now this result holds again asymptotically. However, it is easy to see that *the expected return time does not change at various stages of the walk*, and thus its asymptotic estimate holds even at the beginning of the walk. We conclude that, from the very beginning and on, the expected time between two successive visits of the walk to node i is precisely $\frac{2|E|}{d_i}$.

Suppose now that the input graph G of our randomized algorithm for UNDIRECTED REACHABILITY does have a path from 1 to n, say $[i_0 = 1, i_1, \ldots, i_m = n]$ (if no such path exists, the random walk can never return a false positive...). Since we start from 1, we know that every $\frac{2|E|}{d_1}$ steps we will be returning to 1. So, after an expected number $\frac{d_1}{2}$ of such returns—and therefore an expected number of $|E|$ steps totally—the walk will head in the right direction, to i_1. Now that we are in i_1, we will be returning there on the average every $\frac{2|E|}{d_{i_1}}$ steps, and after an expected number $\frac{d_{i_1}}{2}$ of such returns, or $|E|$ steps, we will

arrive at i_2. And so on. It follows that *after an expected number of fewer than* $|E|n$ *steps we will have arrived at* n. That is, the expected number of steps before the random walk arrives at n is at most $|E|n$.

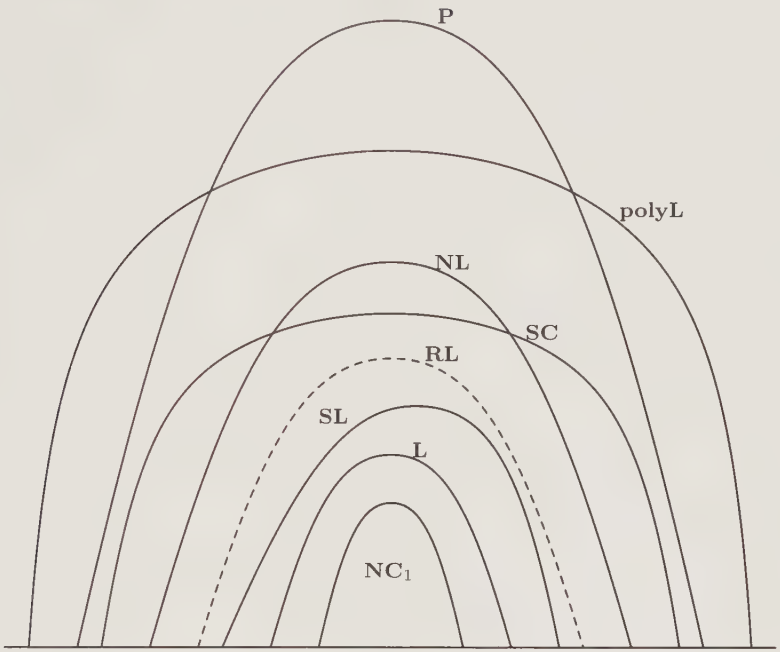
The full randomized algorithm is this:

> Run the random walk from node 1 for $2n|E|$ steps.
> If node n is ever visited, reply "there is a path from 1 to n."
> Otherwise, reply "there is probably no path from 1 to n."

Obviously there are no false positives, and the probability of a false negative is at most $\frac{1}{2}$ (because we run the algorithm for twice the expected time of convergence, recall Lemma 11.2). Finally, it is easy to see that each computation of the algorithm can be implemented in logarithmic space. □

16.4 NOTES, REFERENCES, AND PROBLEMS

16.4.1 Class review:



16.4.2 Theorem 16.1 was proved in

- A. Borodin "On relating time and space to size and depth," *SIAM J. Comp. 6*, pp. 733–744, 1977.

16.4.3 Polylogarithmic space and SC. Logarithmic space may appear a little too restrictive, since for all we know it does not contain "easy" problems like REACHABILITY. One interesting relaxation is *polylogarithmic space*. Define **polyL** to be $\mathbf{SPACE}(\log^k n)$ (remember, this denotes the union over all $k > 0$). Obviously, $\mathbf{NL} \subseteq \mathbf{polyL}$, but it is unclear how **polyL** compares with **P**.

(a) Show that $\mathbf{polyL} \neq \mathbf{P}$. (Can **polyL** have complete sets?)

Since **polyL** is not expected to be contained in **P**, it fails to be a plausible formulation of feasible computation. How can we remedy this? One idea would be to consider the class $\mathbf{polyL} \cap \mathbf{P}$. However, a better and more elegant concept of feasible computation is the class $\mathbf{SC}^{\dagger}$. **SC** is defined as the class of languages, each of which

---

† **SC** stands for "Steve's class;" Nick Pippenger proposed this term to honor Stephen Cook who had first defined and studied this aspect of complexity. Cook had already called "Nick's class," or **NC**, the influential notion of feasibly parallel computation that Pippenger had proposed, recall Section 15.3.

is decided by a deterministic Turing machine that expends *both* polynomial time *and* polylogarithmic space.

(b) Is there a relation between **SC** and $\mathbf{polyL} \cap \mathbf{P}$? To understand the difference of the two classes, show that $\mathbf{NL} \subseteq \mathbf{polyL} \cap \mathbf{P}$, and compare with part (d) below.

We know what the *depth* of a circuit is. It can be defined as follows: Let S_0 denote the set of input gates, and for $j > 0$ let S_j denote the set of all gates that are in no S_i, $i < j$, and all predecessors of which are in some $S_i, i < j$. Now the *depth* of the circuit is the largest j for which S_j is non-empty. The *width* of a circuit is $\max_{j>0} |S_j|$. Notice that taking the maximum over all $j > 0$ allows the width to be less than the number of inputs.

(c) Show that **SC** coincides with the set of languages decided by a uniform family of circuits of polynomial size *and polylogarithmic width.* (Notice the curious *antimnemonic* here: **NC** stands for "shallow circuits," while **SC** for "narrow circuits.")

(d) It is open whether $\mathbf{NL} \subseteq \mathbf{SC}$. Why does it not follow from Savitch's theorem? (How much *time* does the "middle-first search" algorithm in the proof of Savitch's theorem take?)

However, we *do* know now that $\mathbf{RL} \subseteq \mathbf{SC}$. This important result is from

- N. Nisan "RL $\subseteq$ SC," *Proc. 24th ACM Symp. on the Theory of Computing*, pp. 619–623, 1992.

16.4.4 To establish that a problem is **L**-complete we need a weaker version of reduction, one that involves computation even more restrictive than deterministic logarithmic space. An example would be *reductions that can be carried out by* $\mathbf{NC}_1$ *circuits.*

Problem: Show that the reachability problem for *directed trees* is **L**-complete under $\mathbf{NC}_1$ reductions. (For this and other **L**-complete problems see

- N. D. Jones, E. Lien, and W. T. Laaser "New problems complete for deterministic log space," *Math. Systems Theory 10,* pp. 1–17, 1976, and
- S. A. Cook and P. McKenzie "Problems complete for logarithmic space," *J. Algorithms, 8,* pp. 385–394, 1987.

In the literature reductions even weaker than $\mathbf{NC}_1$ are used for this level.)

16.4.5 Alternating Turing machines were introduced and studied in

- A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer "Alternation," *J.ACM, 28,* pp. 114–133, 1981,

where the close relationship between alternating space and exponential time (Corollary 1 to Theorem 16.5), as well as that between alternating time and deterministic space (see Chapter 20) were pointed out. Alternation can be considered as a model of parallelism. In fact, it can be tailored to emulate our uniform circuits (and therefore PRAMS): Suppose that we restrict *both* the time and space of alternating Turing machines (as we did with deterministic Turing machines to define **SC**, recall 16.4.3). In particular, we are interested in alternating Turing machines that use logarithmic space *and* $\log^i n$ time.

Problem: Show that the resulting class is $\mathbf{NC}_i$. (This, as well as a similarly elegant characterization of $\mathbf{AC}_i$ in terms of *total configuration space* and *amount of alternation per computation*, is from

- W. L. Ruzzo "On uniform circuit complexity," *J.CSS, 22*, pp. 365–383, 1981.)

16.4.6 Problem: A *hammer* is a graph with $2n$ nodes, n of which form a clique, while the other n form a path; a node of the clique and an endpoint of the path are connected by an edge.

Show that the random walk algorithm, applied to a hammer, takes an expected $\Omega(n^3)$ steps to reach all nodes of the graph. Hence, the bound in the proof of Theorem 16.6 is asymptotically optimal.

16.4.7 Problem: Consider the directed graph with nodes $\{1, 2, \ldots, n\}$, and edges $\{(i, i+1), (i, 1) : i = 1, \ldots, n-1\}$. How long would the random walk algorithm take to reach node n, if started from node 1?

16.4.8 Universal traversal sequences. Let $G = (V, E)$ be an undirected graph with $V = \{1, 2, \ldots, n\}$. Assume that for each node i we have ordered the edges incident upon i in the sequence $E_i = ([i, j_1], \ldots, [i, j_{k_i}])$ for some $k_i < n$. We can thus think of G as n mappings $G_1, \ldots, G_n$ from V to V, where for $k \leq k_i$ $G_i(k) = j_k$, and for $k > k_i$ $G_i(k) = i$ (that is, we assume that each node has degree n, possibly with several self loops).

Let $U = u_1 u_2 \ldots u_m \in \{0, 1, \ldots, n\}^*$ be a string, G a graph as above, and i a node of G. We define $U(G, i)$ to be the sequence $(i_0, i_1, \ldots, i_m)$ of nodes, where: (a) $i_0 = i$, and (b) for all $j < m$ $i_{j+1} = G_{i_j}(u_m)$. That is, $U(G)$ is the sequence of all nodes visited by the walk that starts at i and follows next the edge out of the current node that is pointed to by the current symbol in U. We say that U *traverses* G if all nodes of G appear in $U(G)$. Finally, U is a *universal traversal sequence for n nodes* if it traverses all connected graphs G with n nodes.

Use a non-constructive probabilistic argument to show that there are universal traversal sequences for n nodes with length $\mathcal{O}(n^3)$.

16.4.9 Theorem 16.6, as well as the previous problem, are from

- R. Aleliunas, R. M. Karp, R. J. Lipton, L. Lovász, and C. Rackoff "Random walks, traversal sequences, and the complexity of maze problems," *Proc. 20th IEEE Symp. on the Foundations of Computer Science*, pp. 218–223, 1979.

16.4.10 Symmetric space. Unidirected reachability seems easier than general reachability because of the convenient *symmetry* of undirected graphs (compare Theorem 16.6 with Theorem 16.2). Is there a way to restrict space-bounded nondeterminism so that it captures precisely this kind of reachability?

(a) Define carefully a variant of the nondeterministic Turing machine for which the "yields in one step" relation among configurations is symmetric. (You may have to define cursors that scan more than one symbol at a time.)

(b) Show that UNDIRECTED REACHABILITY is complete for the class of all languages accepted by machines as in (a) above using only logarithmic space. (This is from

- H. R. Lewis and C. H. Papadimitriou "Symmetric space-bounded computation," *Theor. Comp. Science, 19*, pp. 161–187, 1982.)

Incidentally, it is *not* known whether symmetric space is closed under complement, recall Theorem 7.6. The difficulty is pointed out in

- A. Borodin, S. A. Cook, P. W. Dymond, W. L. Ruzzo, and M. L. Tompa "Two applications of inductive counting for complementation problems," *SIAM J. Comp., 18*, pp. 559–578, 1989.

We *do* know, however, that **SL**, as symmetric logarithmic space is denoted, is weaker than **NL** in at least the following three ways. First, by Theorem 16.6, $\mathbf{SL} \subseteq \mathbf{RL}$. In fact, by a result in the paper just referenced, $\mathbf{SL} \subseteq \mathbf{coRL}$, and so **SL** has *Las Vegas* logarithmic space algorithms (recall the class **ZPP** in Section 11.3); this result combines the techniques in the proofs of Theorems 7.6 and 16.6. Finally, we now know that $\mathbf{SL} \subseteq \mathbf{SPACE}(\log^{\frac{3}{2}} n)$, whereas for **NL** the strongest result we know is Savitch's theorem: $\mathbf{NL} \subseteq \mathbf{SPACE}(\log^2 n)$. That inclusion was proven in

- N. Nisan, E. Szemerédi, A. Wigderson "Undirected connectivity in $\mathcal{O}(\log^{1.5} n)$ space," *Proc. 33rd IEEE Symp. on the Foundations of Computer Science*, pp. 24–29, 1992.

16.4.11 Prove Theorem 16.4 (this is from

- E. Grädel "The expressive power of second-order Horn logic," *Proc. 8th Symp. on Theor. Aspects of Comp. Sci.*, vol. 480 of Lecture Notes in Computer Science, pp. 466–477, 1991.)

PART V

BEYOND NP

If the scope of complexity theory were confined in separating the efficiently solvable problems from the intractable ones, there would probably be little point in studying the classes that lie beyond NP, or their complete problems. However, our purpose here is a little broader than this: We wish to understand the process whereby computational concepts are identified with applications, in terms of reductions and completeness. We feel that we have understood the complexity of a problem only when we have proved it complete for a natural complexity class; but of course, whether a complexity class is natural and important largely depends on its complete problems, and how natural and important they *are. Often research in complexity is led to the definition of an interesting new complexity class by its complete problems, which had resisted precise classification in terms of known complexity classes.*

Besides, complexity often tells us much more about a problem, than just how hard it is to solve. Sometimes it is more useful to look at a complexity result as an allegory *about how* conceptually difficult *the underlying application area may be. After all, if algorithms are often the direct product of mathematical structure, computational complexity must be the manifestation of lack of structure, of mathematical nastiness. From this point of view, in the coming chapters we shall see that playing two-person games is more complex than solving optimization problems, counting combinatorial structures and computing the permanent of a matrix is somewhere in between, decision-making under uncertainty and interactive protocols both are as powerful as games, while succinct input representations make things even harder.*

CHAPTER 17

THE POLYNOMIAL HIERARCHY

Although the complexity classes we shall study now are in one sense byproducts of our definition of NP, they have a remarkable life of their own.

17.1 OPTIMIZATION PROBLEMS

Optimization problems have not been classified in a satisfactory way within the theory of **P** and **NP**; it is these problems that motivate the immediate extensions of this theory beyond **NP**.

Let us take the traveling salesman problem as our working example. In the problem TSP we are given the distance matrix of a set of cities; we want to find the shortest tour of the cities. We have studied the complexity of the TSP within the framework of **P** and **NP** only indirectly: We defined the decision version TSP (D), and proved it **NP**-complete (corollary to Theorem 9.7). For the purpose of understanding better the complexity of the traveling salesman problem, we now introduce two more variants.

EXACT TSP: Given a distance matrix and an integer B, is the length of the shortest tour *equal* to B? Also,

TSP COST: Given a distance matrix, compute the length of the shortest tour.

The four variants can be ordered in "increasing complexity" as follows:

TSP (D); EXACT TSP; TSP COST; TSP.

Each problem in this progression can be reduced to the next. For the last three problems this is trivial; for the first two one has to notice that the reduction in

the corollary to Theorem 9.7 proving that TSP (D) is **NP**-complete can be used to reduce HAMILTON PATH to EXACT TSP (the graph has a Hamilton path if and only if the optimum tour has length *exactly* $n + 1$). And since HAMILTON PATH is **NP**-complete and TSP (D) is in **NP**, we must conclude that there is a reduction from TSP (D) to EXACT TSP.

Actually, *we know that these four problems are polynomially equivalent* (since the first and the last one are, recall Example 10.4). That is, there is a polynomial-time algorithm for one if and only if there is for all four. Admittedly, from the point of view of the practical motivation for complexity theory (namely, to identify problems that are likely to require exponential time) this coarse characterization should be good enough. However, reductions and completeness provide far more refined and interesting categorizations of problems. In this sense, of these four variants of the TSP we know the precise complexity only of the **NP**-complete problem TSP (D). *In this section we shall show that the other three versions of the* TSP *are complete for some very natural extensions of* **NP**.

The Class DP

Is the EXACT TSP in **NP**? Given a distance matrix and the alleged optimum cost B, how can we certify succinctly that the optimum cost is indeed B? The reader is invited to ponder about this question; no obvious solution comes to mind. It would be equally impressive if we could certify that the optimum cost *is not* B; in other words, EXACT TSP does not even appear to be in **coNP**. In fact, the results in this section will suggest that if EXACT TSP is in $\mathbf{NP} \cup \mathbf{coNP}$, this would have truly remarkable consequences; the world of complexity would have to be vastly different than it is currently believed.

However, EXACT TSP is closely related to **NP** and **coNP** in at least one important way: Considered as a language, *it is the intersection of a language in* **NP** (the TSP language) *and one in* **coNP** (the language TSP COMPLEMENT, asking whether the optimum cost is *at least* B). In other words, an input is a "yes" instance of EXACT TSP if and only if it is a "yes" instance of TSP, and a "yes" instance of TSP COMPLEMENT. This calls for a definition:

Definition 17.1: A language L is in the class **DP** if and only if there are two languages $L_1 \in \mathbf{NP}$ and $L_2 \in \mathbf{coNP}$ such that $L = L_1 \cap L_2$. □

We should warn the reader immediately against a quite common misconception: **DP** *is not* $\mathbf{NP} \cap \mathbf{coNP}$[†]. There is a world of difference between these two classes. For one thing, **DP** is not likely to be contained even in $\mathbf{NP} \cup \mathbf{coNP}$, let alone the much more restrictive $\mathbf{NP} \cap \mathbf{coNP}$. The intersection in the definition of $\mathbf{NP} \cap \mathbf{coNP}$ is in the domain of *classes of languages*, not languages as

[†] We mean, these two classes are not *known* or *believed* to be equal. In the absence of a proof that $\mathbf{P} \neq \mathbf{NP}$ one should not be too emphatic about such distinctions.

with **DP**.

For another important difference between **NP** $\cap$ **coNP** and **DP**, the latter is a perfectly syntactic class, and therefore has *complete problems*. Consider for example the following problem:

SAT-UNSAT: Given two Boolean expressions ϕ, ϕ', both in conjunctive normal form with three literals per clause. Is it true that ϕ is satisfiable and ϕ' is not?

Theorem 17.1: SAT-UNSAT is **DP**-complete.

Proof: To show that it is in **DP** we have to exhibit two languages $L_1 \in$ **NP** and $L_2 \in$ **coNP** such that the set of all "yes" instances of SAT-UNSAT is $L_1 \cap L_2$. This is easy: $L_1 = \{(\phi, \phi') : \phi \text{ is satisfiable}\}$ and $L_2 = \{(\phi, \phi') : \phi' \text{ is unsatisfiable}\}$.

To show completeness, let L be any language in **DP**. We have to show that L reduces to SAT-UNSAT. All we know about L is that there are two languages $L_1 \in$ **NP** and $L_2 \in$ **coNP** such that $L = L_1 \cap L_2$. Since SAT is **NP**-complete, we know that there is a reduction R_1 from L_1 to SAT, and a reduction R_2 from the complement of L_2 to SAT. The reduction from L to SAT-UNSAT is this, for any input x:

$$R(x) = (R_1(x), R_2(x)).$$

We have that $R(x)$ is a "yes" instance of SAT-UNSAT if and only if $R_1(x)$ is satisfiable and $R_2(x)$ is not, which is true if and only if $x \in L_1$ and $x \in L_2$, or equivalently $x \in L$. □

As usual, starting from our basic "satisfiability-oriented" complete problem we can show many more **DP**-completeness results:

Theorem 17.2: EXACT TSP is **DP**-complete.

Proof: We already argued that it is in **DP**. To prove completeness, we shall reduce SAT-UNSAT to it. So, let (ϕ, ϕ') be an instance of SAT-UNSAT. We shall use the reduction from 3SAT to HAMILTON PATH (recall the proof of Theorem 9.7) to produce from (ϕ, ϕ') two graphs (G, G'), each of which has a Hamilton path if and only if the corresponding expression is satisfiable. But our construction will be novel in this way: Whether or not the expressions are satisfiable, the graphs G and G' will always contain a *broken Hamilton path*, that is, two node-disjoint paths that cover all nodes.

To this end, we modify slightly each expression so that it has an *almost satisfying* truth assignment, that is, a truth assignment that satisfies all clauses except for one. This is easy to do: We add a new literal, call it z, to all clauses, and add the clause $(\neg z)$. This way, by setting all variables to **true** we satisfy all clauses except for the new one. We then turn the expression into one with three literals per clause by replacing the clause $(x_1 \vee x_2 \vee x_3 \vee z)$, say, by the two clauses $(x_1 \vee x_2 \vee w)$ and $(\neg w \vee x_3 \vee z)$.

If we now perform the reduction in Theorem 9.7 starting from a set of clauses that has such an almost satisfying truth assignment, call it T, it is easy

to see that the resulting graph always has a broken Hamilton path: It starts at node 1, it traverses all variables according to T, and continues to the clauses *except for the one that may be unsatisfied*, where the path is broken once (you may want to examine the "constraint gadget" in Figure 9.6 to verify that it causes at most one such break). The path then continues normally up to node 2.

We shall use this fact to show that SAT-UNSAT can be reduced to EXACT TSP. Given an instance (ϕ, ϕ') of SAT-UNSAT, we apply to both ϕ and ϕ' the reduction to HAMILTON PATH, to obtain two graphs, G and G', respectively, both guaranteed to have broken Hamilton paths. We next combine the two graphs *in a cycle* by identifying node 2 of G with node 1 of G', and vice-versa (Figure 17.1). Let n be the number of nodes in the new graph.

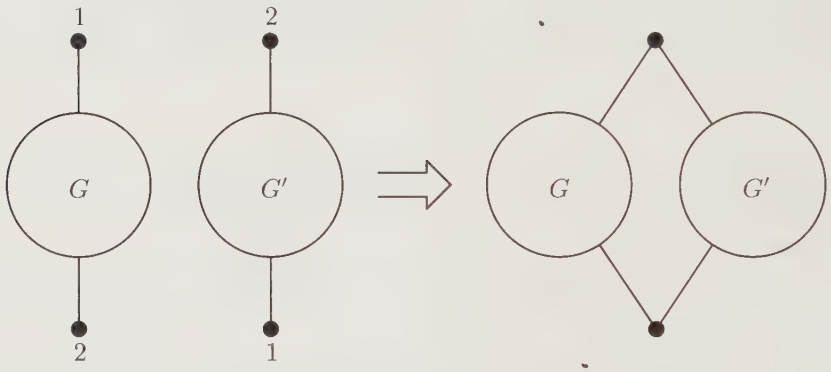


Figure 17-1. Combining G and G'.

We next define distances between the nodes of the combined graph to obtain an instance of the TSP. The distance between nodes i and j are defined as follows: If $[i, j]$ is an edge of either graph G or graph G', then the distance is one. If $[i, j]$ is not an edge, but both i and j are nodes of graph G then its distance is two; all other non-edges have distance 4.

What is the length of the shortest tour of this instance of the traveling salesman problem? Obviously, this depends on whether ϕ and ϕ' are satisfiable or not. If they are both satisfiable, then the optimum cost is n, the number of nodes in the combined graph (there is a Hamilton cycle in the combined graph). If they are both unsatisfiable, then the optimum cost is $n + 3$ (the optimum tour combines the two broken Hamilton paths, and thus both a non-edge of G and a non-edge of G' will have to be used). If ϕ is satisfiable and ϕ' is not, then the optimum cost is $n + 2$ (a non-edge of G' will have to be used, but not of G). And if ϕ is unsatisfiable and ϕ' is satisfiable, then the optimum cost is $n + 1$.

It follows that (ϕ, ϕ') *is a "yes" instance of* SAT-UNSAT *if and only if the optimum cost is* $n + 2$. Taking B to be equal to this number completes our reduction from SAT-UNSAT to EXACT TSP. □

The "exact cost" versions of all **NP**-complete optimization problems that we have seen (INDEPENDENT SET, KNAPSACK, MAX-CUT, MAX SAT, to name a few) can be shown **DP**-complete, each by a different trick that combines two instances, and forces the optimum cost to precisely reflect the status of the two expressions. So, **DP** appears to be the natural niche of the "exact cost" aspect of optimization problems.

But **DP** is much richer than this. For example, besides SAT-UNSAT, there are two more satisfiability-related problems in **DP**:

CRITICAL SAT: Given a Boolean expression ϕ, is it true that ϕ is unsatisfiable, but deleting any clause makes it satisfiable?

UNIQUE SAT: Given a Boolean formula ϕ, is it true that it has a unique satisfying truth assignment?

CRITICAL SAT exemplifies an important and novel genre of problems, those asking whether the input is *critical* with respect to a given property, that is, it has the property but its slightest unfavorable perturbation does not. Other examples:

CRITICAL HAMILTON PATH: Given a graph, is it true that it has no Hamilton path, but addition of any edge creates a Hamilton path?

CRITICAL 3-COLORABILITY: Given a graph, is it true that it is not 3-colorable, but deletion of any node makes it 3-colorable?

All three "critical" problems are known to be **DP**-complete. On the other hand UNIQUE SAT, and many other problems asking whether a given instance has a unique solution, are simply not known to be in any weaker class. They are not known (or believed) to be **DP**-complete (see the references). Incidentally, UNIQUE SAT should not be confused with the class **UP** of unambiguous nondeterministic computations (recall Section 12.2). The two address very different aspects of unique solutions in decision problems: UNIQUE SAT is about *determining* whether the solution exists and is unique; **UP** concerns the computational power of instances that are guaranteed either to have a unique solution or no solution. The satisfiability problem for **UP**, call it UNAMBIGUOUS SAT, would be the following: Given a Boolean expression that is known to have at most one satisfying truth assignment, does it have one? This is a completely different problem than UNIQUE SAT.

The Classes $\mathbf{P}^{\mathbf{NP}}$ and $\mathbf{FP}^{\mathbf{NP}}$

One can look at **DP** as the class of all languages that can be decided by an oracle machine (recall Section 14.3) of a very special nature: The machine makes two

queries to a SAT oracle, and then accepts if and only if the first answer was "yes" and the second was "no." Obviously, one can generalize this to situations where the acceptance pattern is *any fixed Boolean expression* (in the case of **DP**, for example, the expression is $x_1 \wedge \neg x_2$, see the references).

But the more interesting generalization is to allow *any polynomial number of queries*, and in fact queries computed *adaptively*, based on the answers of previous queries. This way we arrive at the class $\mathbf{P}^{\mathrm{SAT}}$, the class of all languages decided by polynomial-time oracle machines with a SAT oracle. Since SAT is **NP**-complete, instead of it we could use as an oracle any language in **NP**—this is why we can equivalently write $\mathbf{P}^{\mathrm{SAT}}$ as $\mathbf{P}^{\mathbf{NP}}$. Yet another name for this class is $\Delta_2\mathbf{P}$; this name identifies $\mathbf{P}^{\mathbf{NP}}$ as one of the first levels of an important progression of classes, discussed in the next section.

Having defined $\mathbf{P}^{\mathbf{NP}}$, we can now define its corresponding class of *functions* $\mathbf{FP}^{\mathbf{NP}}$ (recall **FP** and **FNP** in Chapter 10). That is, $\mathbf{FP}^{\mathbf{NP}}$ is the class of all functions from strings to strings that can be computed by a polynomial-time Turing machine with a SAT oracle. In fact, we shall be much more interested in $\mathbf{FP}^{\mathbf{NP}}$ than in $\mathbf{P}^{\mathbf{NP}}$, because the former class happens to have many natural complete problems, *including many important optimization problems.* For example, $\mathbf{FP}^{\mathbf{NP}}$ finally provides the sought precise characterization of the complexity of the TSP.

There are several natural $\mathbf{FP}^{\mathbf{NP}}$-complete problems. The version of satisfiability appropriate for this level is the following:

MAX-WEIGHT SAT: Given a set of clauses, each with an integer weight, find the truth assignment that satisfies a set of clauses with the most total weight.

But our reductions this time will start with a problem that is even closer to computation than satisfiability:

MAX OUTPUT: We are given a nondeterministic Turing machine N and its input 1^n. N is such that, on input 1^n, and for any sequence of nondeterministic choices, it halts after $\mathcal{O}(n)$ steps with a binary string of length n on its output string. We are asked to determine *the largest output*, considered as a binary integer, of any computation of N on 1^n.

Theorem 17.3: MAX OUTPUT is $\mathbf{FP}^{\mathbf{NP}}$-complete.

Proof: Let us first point out that MAX OUTPUT, along with any optimization problem whose decision version is in **NP**, is in $\mathbf{FP}^{\mathbf{NP}}$. The algorithm is essentially the one used for the TSP (Example 10.4): Given N and 1^n, we repeatedly ask whether there is a sequence of nondeterministic choices leading to an output larger than an integer x. We repeat this for various integers x, converging to the value of the optimum by binary search. Each such question can be answered in **NP**, and hence the resulting algorithm establishes that MAX OUTPUT is in $\mathbf{FP}^{\mathbf{NP}}$. (Incidentally, notice that the binary search algorithm is *adaptive*, in that it makes nontrivial use of the answers to previous queries in order to

construct the next query; in some sense, the result being proved suggests that *binary seach is the most general way of doing this.*)

Suppose then that F is a function from strings to strings in $\mathbf{FP}^{\mathbf{NP}}$. That is, there is a polynomial-time oracle machine $M^?$ such that for all inputs x $M^{\mathrm{SAT}}(x) = F(x)$. We shall describe a reduction from F to MAX OUTPUT. Since this is a reduction between function problems, what is required is two functions R and S such that (a) R and S are computable in logarithmic space; (b) for any string x $R(x)$ is an instance of MAX OUTPUT; and (c) S applied to the maximum output of $R(x)$ returns $F(x)$, the value of the function on the original input x.

Given x, we shall first describe the R part of the reduction, that is, how to construct machine N and its input 1^n. To start, define $n = p^2(|x|)$, where $p(\cdot)$ is the polynomial bound of M^{SAT}—this will give N plenty of time to simulate M^{SAT}. We describe N informally, like any other nondeterministic Turing machine; it will be clear that its transition relation can be constructed in logarithmic space, starting from x. N on input 1^n first generates x on a string (this is the only place in the construction where x is needed), and then it simulates M on input x. The simulation is very easy and deterministic, *except for the query steps of* M^{SAT}.

Suppose that M^{SAT} arrives at its first query step, asking whether some Boolean expression ϕ_1 is satisfiable. N simulates this by nondeterministically guessing the answer z_1 to this query—z_1 is 1 if ϕ is satisfiable, 0 otherwise. If $z_1 = 0$, then N simply continues its simulation of M^{SAT}, naturally from state q_{NO}. But if $z_1 = 1$, then N goes on to guess a satisfying truth assignment T_1 for ϕ_1, and check that indeed T_1 satisfies ϕ_1. If the test succeeds, then N goes on to simulate M^{SAT} from state q_{YES}. But if the test fails, then N writes the smallest possible output, 0^n, and halts; we call this an *unsuccessful computation.*

N continues this way to simulate M^{SAT} on input x, using its nondeterminism to guess the answers $z_i, i = 1, \ldots$ of all queries. When M^{SAT} would halt, N outputs *the bit string* $z_1 z_2 \ldots$ *of the alleged answers to the queries,* followed by enough zeros to bring the total length of the output up to n, followed by the output of M^{SAT} (needed for the S part). This is a *successful computation.*

Many of the successful computations of N will be *erroneous simulations of* M^{SAT}, in the sense that maybe a query ϕ_j was satisfiable, and still $z_j = 0$—every successful computation will be correct about $z_j = 1$. But we claim that *the successful computation that outputs the largest integer does correspond to a correct simulation.* The reason is simple: Suppose that in the successful computation which leads to the largest output, we have $z_j = 0$ for some j, while ϕ_j was satisfiable —say by truth assignment T_j. Take the smallest such j (that is, the earliest such mistake). But then there is another successful computation of N, which is identical to the present one up to the jth query step, at which point it guesses $z_j = 1$, and then goes on to correctly guess

the truth assignment T_j, check it, and continue successfully to the end. But the output of this other computation agrees with the present one in the first $j-1$ bits, and has a 1 in its jth position. Hence it represents a larger number, contradicting the maximality of the present computation. It follows that the computation of N with the largest output does indeed correspond to a correct simulation of M.

To summarize the structure of N, it has $|x|$ states for writing x on its string, and uses its $p^2(|x|)$-long input as an alarm clock. The rest of its transition relation reflects the transition function of $M^?$, with the exception of the query state, which is simulated by a simple nondeterministic routine. It should be clear that N can be constructed in logarithmic space. As for the S part of the reduction, $F(x)$ can be simply read off the end of the largest output of N. □

Theorem 17.4: MAX-WEIGHT SAT is $\mathbf{FP}^{\mathbf{NP}}$-complete.

Proof: The problem is in $\mathbf{FP}^{\mathbf{NP}}$: By binary search, using a SAT oracle, we can find the largest possible total weight of satisfied clauses, and then, by setting the variables one-by-one, the truth assignment that achieves it.

We must now reduce MAX OUTPUT to MAX-WEIGHT SAT. As in the reduction in Cook's theorem (Theorem 8.2), starting from the nondeterministic machine N and its input 1^n we can construct a Boolean expression $\phi(N,n)$ such that any satisfying truth assignment of $\phi(N,n)$ corresponds to a legal computation of N on input 1^n. All clauses in $\phi(N,n)$ are given a huge weight, say 2^n, so that any truth assignment that aspires to be optimum must satisfy all clauses of $\phi(N,n)$.

We now add some more clauses to $\phi(N,n)$. Recall that in $\phi(N,n)$ there are variables corresponding to the symbol contained in every position of every string of N, at every step. So, there are n variables, call them $y_1,\ldots,y_n$, corresponding to the bits of the output string at halting. *We add to our instance of* MAX-WEIGHT SAT *the one-literal clauses* $(y_i) : i = 1,\ldots,n$, *where clause* (y_i) *has weight* 2^{n-i}. It is easy to see that, because of these new clauses, and their weights that are the right powers of two, the optimum truth assignment must now not represent just any legal computation of N on input 1^n, but it must represent the computation that produces the output with the largest possible binary integer value. Finally, for the S part of the reduction, from the optimum truth assignment of the resulting expression (in fact, even from the optimum weight alone!) we can easily recover the optimum output of N. □

We can now proceed to the main result of this section:

Theorem 17.5: TSP is $\mathbf{FP}^{\mathbf{NP}}$-complete.

Proof: We know that TSP is in $\mathbf{FP}^{\mathbf{NP}}$ (Example 10.4) To prove completeness, we shall reduce MAX-WEIGHT SAT to it. Given any set of clauses $C_1,\ldots,C_m$ on n variables $x_1,\ldots,x_n$, with weights $w_1,\ldots,w_m$, we shall construct an instance

of the TSP such that the optimum truth assignment of the set of clauses can be easily recovered from the optimum tour.

The TSP instance will be given as usual in terms of a graph. All distances that do not correspond to edges in the graph are prohibitively large, say $W = \sum_{i=1}^{m} w_i$. The graph is a variant of that used in the **NP**-completeness proof of the Hamilton path problem (see Figure 17.2, and compare with the proof of Theorem 9.7). There are "choice" gadgets for the variables connected *in tandem* as before, but the "constraint" gadgets for the clauses are now different: Each constraint gadget consists of *four parallel edges*, three corresponding to the literals of the clause (so that the tour will traverse one of the **true** literals in the clause), plus an extra parallel edge functioning as an "emergency exit:" If the clause is unsatisfied and has no **true** literals, then the three parallel edges will not be available, and the emergency exit must be taken. *All edges of the graph have length* 0 *except for the emergency exits for the clauses, whose length is the weight of the corresponding clause.* This way, each time an emergency edge is taken, the lost weight of the corresponding clause is accurately represented in the cost of the tour.

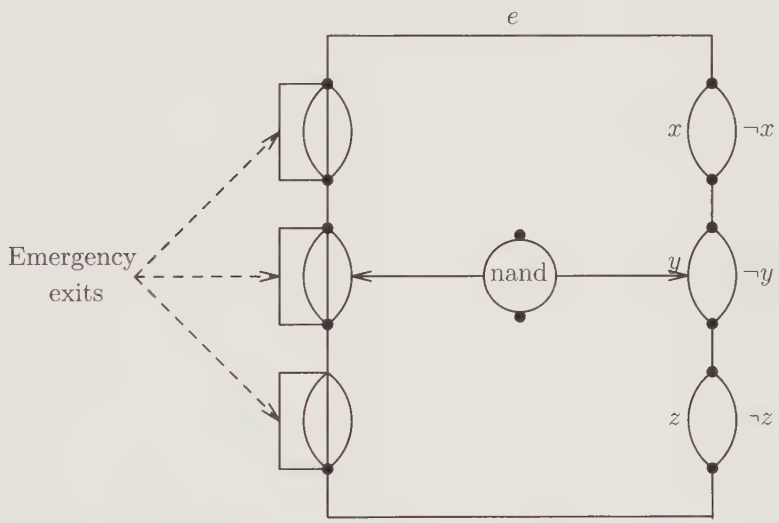


Figure 17-2. The overall construction.

What remains is perhaps the more subtle part of the construction, the "consistency" gadget. Because of the new constraint gadget based on three parallel edges (which is in some sense the "dual" of the triangle we used in Figure 9.6), we must connect literal occurrences with the *opposite literal* in the

choice gadget, not with the same literal as before. More importantly, we must allow for an edge corresponding to an occurrence of a **true** literal *not* to be traversed (in the case that there are two or three **true** literals in the clause). As a result, the "exclusive or" gadget in Figure 9.5 is not appropriate. We must design a "nand" gadget, allowing for the possibility that neither edge is traversed. Such a gadget would connect each literal edge of each clause with the opposite literal in the corresponding choice gadget, ensuring that, once a choice is made, the opposite literals cannot be traversed by the Hamilton cycle.

Our nand gadget is rather complicated (it has 36 nodes!), but the idea in designing it is quite simple: After all, a nand gadget is nothing else but an exclusive-or gadget, which also has the additional option of being "turned off," left untraversed. We can achieve this effect by using the "diamond gadget" shown in Figure 17.3. This graph has the following interesting property, easily checked with a little experimentation: Suppose that it is part of a graph so that, as usual, only the black nodes have edges going to the rest of the graph. Then, it can be traversed by a Hamilton cycle only in one of the two ways shown in the figure: Either "from North to South," or "from East to West." In other words, if a Hamilton cycle enters the graph from any one of the four black nodes, it will have to traverse the whole graph and exit from the opposite node.

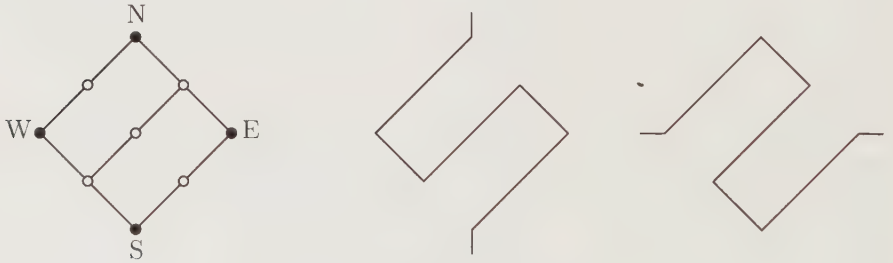


Figure 17-3. The diamond.

Our nand gadget is nothing else but our exclusive-or gadget of Figure 9.5, *only with its four vertical paths of length two replaced each by the diamond gadget* as shown in Figure 17.4(a). It is easy to see that, with this replacement, the overall graph functions exactly as before, as an exclusive-or between its upper and lower edge. The point is that now *we can use the East-West endpoints to turn off the device at will*, by taking the horizontal path shown in the figure. We shall represent the nand gadget as an exclusive-or gadget with an extra path which, if traversed, can leave the rest of the device untraversed, and thus "turned off" (Figure 17.4(b)).

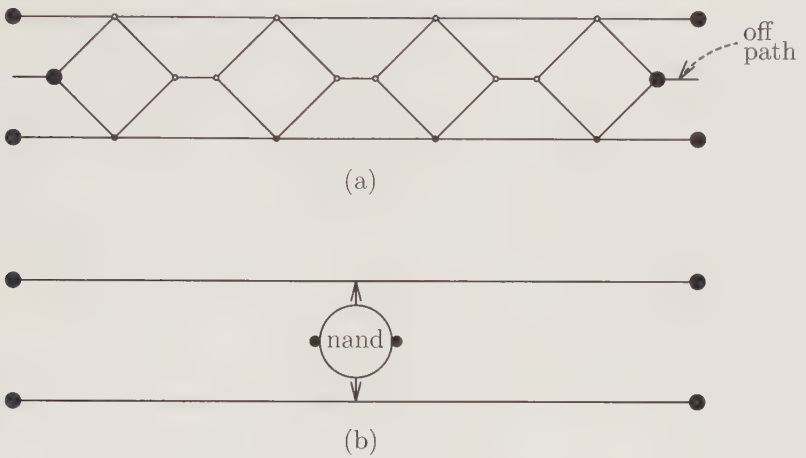


Figure 17-4. The nand gadget.

We now have to modify slightly the "constraint" gadgets in a way that allows just one of the possibly two or three parallel edges corresponding to **true** literals in a clause to be traversed. Recall that each nand gadget corresponds to an occurrence of a literal in a clause. Order arbitrarily the three literals of the clause as *first*, *second*, and *third*. The parallel path corresponding to the first literal now starts by another choice (see Figure 17.5), where the choice is between (1) turning off the nand gate of the second literal (if that literal also happens to be **true**), and (2) not turning it off. Then the path continues with another choice, that of turning off the third literal, in case it is also **true**, or not turning it off. The path corresponding to the second literal has only one choice, between turning off the third literal or not. The third literal has no such choices. In other words, we have given the three literals *priorities:* If the first literal is **true**, then it is traversed and must turn off any of the other literals that may also be **true**. Failing this, if the second literal is **true**, then it must be traversed and possibly turn off the third literal, if **true**. Finally, if only the third literal is **true**, then it must be traversed. And if no literal is T, then the emergency exit must be taken.

The construction is now complete. To review it (see Figure 17.2), we start with a choice for each variable, then four parallel paths for each clause, with extra choices for each of the first two paths to turn off the exclusive-or of the subsequent paths, and finally the cycle is closed. Each literal occurrence edge is connected with the opposite literal in the choice gadget of the corresponding variable by a nand gadget. The emergency edge corresponding to clause C_i has length w_i, all other edges have length zero, while the length of all non-edges is

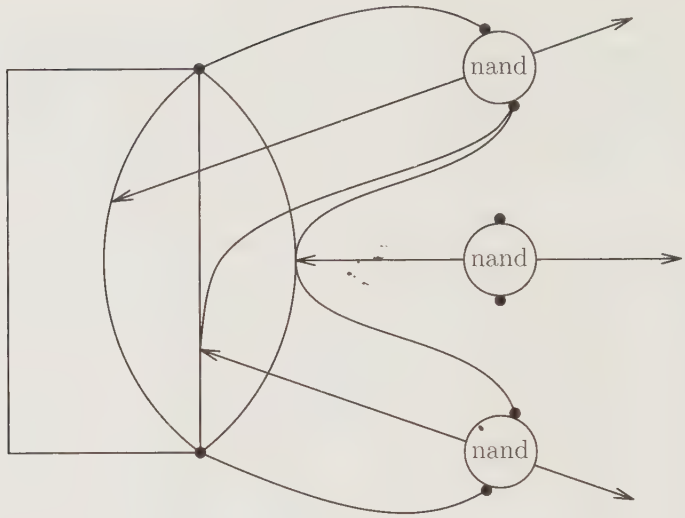


Figure 17-5. The clause gadget.

prohibitively large, say W, the sum of all weights.

Consider now the optimum traveling salesman tour of this instance. Obviously, no non-edges are traversed, and thus the tour is in fact a Hamilton cycle of the graph (and so our gadgets come into play). The tour must traverse the choices for the variables, thus defining a truth assignment, call it T. It then traverses the choices for the nand gates, turning some of them on and some of them off. It finally traverses the clause part. For each clause it must be the case that the tour traverses exactly one of the four parallel edges. This edge can be either a literal that is **true** in T, or the emergency edge. All nand gadgets corresponding to traversed literal occurrences must have been turned "on," while all corresponding to **false** or untraversed **true** literal occurrences must be "off." Finally, the tour is closed, at a total cost equal to the *sum of the weights of the clauses not satisfied by* T, that is, W minus the total weight of T. It follows that *the minimum-length tour corresponds to the maximum-weight truth assignment*, and the proof is complete. □

Corollary: TSP COST is $\mathbf{FP}^{\mathbf{NP}}$-complete.

Proof: Consider the variant of the MAX-WEIGHT SAT problem in which we only return the optimum weight, rather than the optimum truth assignment. It is easy to see that this problem $\mathbf{FP}^{\mathbf{NP}}$-complete; the reduction is essentially the same as that in the proof of Theorem 17.4. Finally, the proof of Theorem 17.5 establishes that this variant of MAX-WEIGHT SAT can be reduced to TSP COST. □

The Class $\mathbf{P}^{\mathbf{NP}[\log n]}$

Many other optimization problems are known to be $\mathbf{FP}^{\mathbf{NP}}$-complete: The (full optimization version of) KNAPSACK, the *weighted versions* of MAX-CUT and BISECTION WIDTH, and so on. Conspicuously absent from this list are problems whose cost is polynomially large and hence has *logarithmically many bits*, such as CLIQUE, UNARY TSP (the TSP with distances written in unary) and the unweighted versions of MAX SAT, MAX-CUT, and BISECTION WIDTH.

And there is a reason for this. Consider for example the problem

CLIQUE SIZE: Given a graph, determine the size of its largest clique.

The binary search algorithm that proves that CLIQUE SIZE is in $\mathbf{FP}^{\mathbf{NP}}$ asks only logarithmically many adaptive **NP** queries —the exact value that must be determined is between one and n, the number of nodes of the given graph, and so binary search takes $\log n$ queries to converge to the true value. Alternatively, we can think of an oracle algorithm for CLIQUE SIZE that asks polynomially many queries (i.e., whether the maximum clique is larger than k, for all values of k from 1 to n); but the queries here *are not adaptive*, they do not depend at all on the answers of previous queries. In either case, the oracle algorithm for CLIQUE SIZE does not make full use of the polynomially many adaptive queries at its disposal (we later show that, quite remarkably, these two kinds of restrictions lead to the same class). Hence, CLIQUE SIZE, and the other optimization problems with polynomially large cost, must belong in a weaker complexity class.

And they do. Let us define $\mathbf{P}^{\mathbf{NP}[\log n]}$ to be the the class of all languages decided by a polynomial-time oracle machine which on input x *asks a total of* $\mathcal{O}(\log |x|)$ SAT *queries*. $\mathbf{FP}^{\mathbf{NP}[\log n]}$ is the corresponding class of functions.

Theorem 17.6: CLIQUE SIZE is $\mathbf{FP}^{\mathbf{NP}[\log n]}$-complete.

Proof: The proof mimics our argument that led to the $\mathbf{FP}^{\mathbf{NP}}$-completeness of TSP. We first show that the problem MAX OUTPUT$[\log n]$, the version of MAX OUTPUT in which the output has $\log n$, not n, bits, is $\mathbf{FP}^{\mathbf{NP}[\log n]}$-complete; the proof is completely analogous to that of Theorem 17.3. We then reduce MAX OUTPUT$[\log n]$ to MAX SAT SIZE (the version of MAX SAT in which the maximum number of satisfied clauses is sought). The idea here is that, since the output of the machine has logarithmically many bits, the weights needed in the proof of Theorem 17.4 are polynomial in n, and hence they can be simulated by *multiple copies of the same clause*. Finally, MAX SAT SIZE is reduced to CLIQUE SIZE by the usual reduction (via INDEPENDENT SET, recall Theorem 9.4 and its corollaries). □

Similarly, the other optimization problems with polynomial-size cost mentioned above can be shown $\mathbf{FP}^{\mathbf{NP}[\log n]}$-complete.

But what about the other restriction on $\mathbf{FP}^{\mathbf{NP}}$, in which the oracle machine

must decide which queries to ask *non-adaptively*, before it knows the answer to any query? Define $\mathbf{P}_{\parallel}^{\mathbf{NP}}$ (for an oracle machine that asks its queries *in parallel*, that is) to be the class of all languages that can be decided by an oracle machine operating as follows: On input x, the machine computes in polynomial time a polynomial number of instances of SAT (or any other problem in **NP**), and receives the correct answers. Based on these answers, the machine decides whether $x \in L$ in polynomial time.

Theorem 17.7: $\mathbf{P}_{\parallel}^{\mathbf{NP}} = \mathbf{P}^{\mathbf{NP}[\log n]}$.

Proof: To show that $\mathbf{P}^{\mathbf{NP}[\log n]} \subseteq \mathbf{P}_{\parallel}^{\mathbf{NP}}$, consider a machine that uses at most $\mathcal{O}(\log n)$ adaptive **NP** queries. When the first query is asked, there are two possibilities, one for each possible answer. For each of these two possibilities there is a next query to be asked, and two possible answers for each. It is easy to see that overall there are $2^{k \log n} = \mathcal{O}(n^k)$ queries that can be possibly asked during the computation. To simulate this machine by a non-adaptive oracle machine, we first compute all $\mathcal{O}(n^k)$ possible queries, find the answers to all of them, and from that we easily determine the correct path to be followed and answer given.

For the other direction, suppose that we have a language decidable by polynomially many non-adaptive SAT queries. We can decide this language with logarithmically many adaptive **NP** queries, as follows: First, in $\mathcal{O}(\log n)$ **NP** queries we determine (by binary search) *the precise number of "yes" answers to the non-adaptive queries.* Notice that each question in this binary search, asking whether the given set of Boolean expressions has satisfying truth assignments for at least k of them, is itself an **NP** query—the k satisfying truth assignments, together with an indication of which expression is satisfied by each, comprise an adequate certificate. Once the exact number k of "yes" answers is known, we ask the last query: "Do there exist k satisfying truth assignments for k of the expressions such that, if all other expressions were unsatisfiable (which we *know* they must be...) the oracle machine would end up accepting?" □

17.2 THE POLYNOMIAL HIERARCHY

Now that we have defined $\mathbf{P}^{\mathbf{NP}}$ we find ourselves in a familiar position: We have defined an important deterministic complexity class (it *is* deterministic, since the oracle machines in terms of which it is defined are deterministic), and we are tempted to consider *the corresponding nondeterministic class*, $\mathbf{NP}^{\mathbf{NP}}$ Naturally, this class most likely will not be closed under complement, and hence we should also consider oracle machines that use *that class.* And so on:

Definition 17.2: The *polynomial hierarchy* is the following sequence of classes: First, $\Delta_0\mathbf{P} = \Sigma_0\mathbf{P} = \Pi_0\mathbf{P} = \mathbf{P}$; and for all $i \geq 0$

$\Delta_{i+1}\mathbf{P} = \mathbf{P}^{\Sigma_i\mathbf{P}}$
$\Sigma_{i+1}\mathbf{P} = \mathbf{NP}^{\Sigma_i\mathbf{P}}$
$\Pi_{i+1}\mathbf{P} = \mathbf{coNP}^{\Sigma_i\mathbf{P}}$.
We also define the *cumulative polynomial hierarchy* to be the class $\mathbf{PH} = \bigcup_{i\geq 0}\Sigma_i\mathbf{P}$. □

Since $\Sigma_0\mathbf{P} = \mathbf{P}$ does not help polynomial-time oracle machines, the first level of this hierarchy makes up our familiar important complexity classes: $\Delta_1\mathbf{P} = \mathbf{P}$, $\Sigma_1\mathbf{P} = \mathbf{NP}$, $\Pi_1\mathbf{P} = \mathbf{coNP}$. The second level starts with the class $\Delta_2\mathbf{P} = \mathbf{P}^{\mathbf{NP}}$ studied in the previous section, and continues with $\Sigma_2\mathbf{P} = \mathbf{NP}^{\mathbf{NP}}$, and its complement $\Pi_2\mathbf{P} = \mathbf{coNP}^{\mathbf{NP}}$. As with the first level, there is every reason to believe that all three classes are distinct. The same holds for the third level, and so on. Naturally, the three classes at each level are related by the same inclusions that we know about **P**, **NP**, and **coNP**. Also, each class at each level includes all classes at previous levels.

In order to show that a problem is in **NP** we are more likely to argue in terms of "certificates" or "witnesses," rather than in terms of nondeterministic Turing machines. We have found it simple and convenient to use the characterization of **NP** in terms of polynomially balanced relations (Proposition 9.1). In the polynomial hierarchy with its complex recursive definition such conceptual simplification is even more welcome, almost essential. We prove below a direct generalization of Proposition 9.1 for the polynomial hierarchy.

Theorem 17.8: Let L be a language, and $i \geq 1$. $L \in \Sigma_i\mathbf{P}$ if and only if there is a polynomially balanced relation R such that the language $\{x; y : (x, y) \in R\}$ is in $\Pi_{i-1}\mathbf{P}$ and

$$L = \{x : \text{ there is a } y \text{ such that } (x, y) \in R\}.$$

Proof: By induction on i. For $i = 1$, the statement is exactly Proposition 9.1. So suppose that $i > 1$, and such a relation R exists. We must show that $L \in \Sigma_i\mathbf{P}$. That is, we must describe a nondeterministic polynomial-time oracle machine, with a language in $\Sigma_{i-1}\mathbf{P}$ as an oracle, that decides L. This is easy: The nondeterministic machine on input x simply guesses an appropriate y, and asks a $\Sigma_{i-1}\mathbf{P}$ oracle whether $(x, y) \in R$ (more correctly, since R is a $\Pi_{i-1}\mathbf{P}$ relation, whether $(x, y) \notin R$).

Conversely, suppose that $L \in \Sigma_i\mathbf{P}$. We must show that an appropriate relation R exists. What we know is that L can be decided by a polynomial-time nondeterministic Turing machine $M^?$ using as an oracle a language $K \in \Sigma_{i-1}\mathbf{P}$. Since $K \in \Sigma_{i-1}\mathbf{P}$, by induction there is a relation S recognizable in $\Pi_{i-2}\mathbf{P}$ such that $z \in K$ if and only if there is a w with $(z, w) \in S$.

We must describe a polynomially balanced, polynomially decidable relation R for L; that is, a succinct certificate for each $x \in L$. We know that $x \in L$ if and only if there is a correct, accepting computation of M^K on x. The certificate

of x will be a string y recording such a computation of M^K (compare with the proof of Proposition 9.1). But recall that M^K is now an oracle machine with an oracle $K \in \Sigma_{i-1}\mathbf{P}$, and thus several of its steps will be queries to K. Some of these steps will have "yes" answers, and some "no" answers. *For each "yes" query z_i, our certificate y also includes z_i's own certificate w_i* such that $(z_i, w_i) \in S$. This is the definition of R: $(x, y) \in R$ if and only if y records an accepting computation of $M^?$ on x, together with a certificate w_i for each "yes" query z_i in the computation.

We claim that checking whether $(x, y) \in R$ can be done in $\Pi_{i-1}\mathbf{P}$. First, we must check whether all steps of $M^?$ are legal; but this can be done in deterministic polynomial time. Then we must check for polynomially many pairs (z_i, w_i) whether $(z_i, w_i) \in S$; but this can be done in $\Pi_{i-2}\mathbf{P}$, and thus certainly in $\Pi_{i-1}\mathbf{P}$. Finally, for all "no" queries z_i' we must check that indeed $z_i' \notin K$. *But since $K \in \Sigma_{i-1}\mathbf{P}$, this is another $\Pi_{i-1}\mathbf{P}$ question.* Thus $(x, y) \in R$ if and only if several $\Pi_{i-1}\mathbf{P}$ queries all have answers "yes," and it is easy to see that this can be done in a single $\Pi_{i-1}\mathbf{P}$ computation. □

The "dual" result for $\Pi_i\mathbf{P}$ is this:

Corollary 1: Let L be a language, and $i \geq 1$. $L \in \Pi_i\mathbf{P}$ if and only if there is a polynomially balanced binary R such that the language $\{x; y : (x, y) \in R\}$ is in $\Sigma_{i-1}\mathbf{P}$ and

$$L = \{x : \text{ for all } y \text{ with } |y| \leq |x|^k, (x, y) \in R\}.$$

Proof: Just recall that $\Pi_i\mathbf{P}$ is precisely $\mathbf{co}\Sigma_i\mathbf{P}$. □

Notice that in the description of L in Corollary 1 we must explicitly state for the universally quantified string y the bound $|y| \leq |x|^k$. Since R is known to be polynomially balanced, this constraint is, in this context, superfluous, and will be omitted. Also, we shall use quantifiers such as $\forall x$ and $\exists y$ in the descriptions of languages such as the one displayed in Corollary 2 below. This will help bring out the elegant mathematical structure of these descriptions, as well as their affinity with logic.

In order to get rid of the recursion in Theorem 17.8, let us call a relation $R \subseteq (\Sigma^*)^{i+1}$ polynomially balanced if, whenever $(x, y_1, \ldots, y_i) \in R$, we have that $|y_1|, \ldots, |y_i| \leq |x|^k$ for some k.

Corollary 2: Let L be a language, and $i \geq 1$. $L \in \Sigma_i\mathbf{P}$ if and only if there is a polynomially balanced, polynomial-time decidable $(i+1)$-ary relation R such that

$$L = \{x : \exists y_1 \forall y_2 \exists y_3 \ldots Q y_i \text{ such that } (x, y_1, \ldots, y_i) \in R\}$$

where the ith quantifier Q is "for all" if i is even, and "there is" if i is odd.

Proof: Repeatedly replace languages in $\Pi_j\mathbf{P}$ or $\Sigma_j\mathbf{P}$ by their certificate forms as in Theorem 17.8 and its Corollary 1. □

Using these characterizations we can prove the basic fact concerning the polynomial hierarchy: As it is built by patiently adding layer after layer, always using the previous layer as an oracle for defining the next, the resulting structure is extremely fragile and delicate. Any jitter, at any level, has disastrous consequences further up:

Theorem 17.9: If for some $i \geq 1$ $\Sigma_i\mathbf{P} = \Pi_i\mathbf{P}$, then for all $j > i$ $\Sigma_j\mathbf{P} = \Pi_j\mathbf{P} = \Delta_j\mathbf{P} = \Sigma_i\mathbf{P}$.

Proof: It suffices to show that $\Sigma_i\mathbf{P} = \Pi_i\mathbf{P}$ implies $\Sigma_{i+1}\mathbf{P} = \Sigma_i\mathbf{P}$. So, consider a language $L \in \Sigma_{i+1}\mathbf{P}$. By Theorem 17.8 there is a relation R in $\Pi_i\mathbf{P}$ with $L = \{x : \text{ there is a } y \text{ such that } (x, y) \in R\}$. But since $\Pi_i\mathbf{P} = \Sigma_i\mathbf{P}$, R is in $\Sigma_i\mathbf{P}$. That is, $(x, y) \in R$ if and only if there is a z such that $(x, y, z) \in S$ for some relation $S \in \Pi_{i-1}\mathbf{P}$. Thus $x \in L$ if and only if there is a string $y; z$ such that $(x, y, z) \in S$, where $S \in \Pi_{i-1}\mathbf{P}$. But this means that $L \in \Sigma_i\mathbf{P}$. □

The statements of many results in complexity theory end like that of Theorem 17.9: "then for all $j > i$ $\Sigma_j\mathbf{P} = \Pi_j\mathbf{P} = \Delta_j\mathbf{P} = \Sigma_i\mathbf{P}$." This conclusion is usually abbreviated "then the polynomial hierarchy collapses to the ith level." For example:

Corollary: If $\mathbf{P} = \mathbf{NP}$, or even if $\mathbf{NP} = \mathbf{coNP}$, the polynomial hierarchy collapses to the first level. □

The last corollary makes one thing abundantly clear: In the absence of a proof that $\mathbf{P} \neq \mathbf{NP}$, there is no hope of proving that the polynomial "hierarchy" is indeed a hierarchy of classes each properly containing the next (although, once again, we strongly believe that it is). Still, the polynomial hierarchy is interesting for several reasons. First it is the polynomial analog of an important (provable) hierarchy of "more and more undecidable problems," the *arithmetic* or *Kleene hierarchy* (recall Problem 3.4.9). Second, its various levels do contain some, even though not very many, interesting and natural problems; some of them are complete. For example, consider the following decision problem:

MINIMUM CIRCUIT: Given a Boolean circuit C, is it true that there is no circuit with fewer gates that computes the same Boolean function?

MINIMUM CIRCUIT is in $\Pi_2\mathbf{P}$, and not known to be in any class below that. To see that it is in $\Pi_2\mathbf{P}$, notice that C is a "yes" instance if and only if *for all circuits C' with fewer gates there is an input x for which $C(x) \neq C'(x)$*. Then use Corollary 2, noting that the last inequality can be checked in polynomial time.

It is open whether MINIMUM CIRCUIT is $\Pi_2\mathbf{P}$-complete. Fortunately, and as usual, for every $i \geq 1$ there is a version of satisfiability very appropriate for the corresponding level of the hierarchy:

QSAT$_i$ (for *quantified satisfiability with i alternations of quantifiers*): Given a Boolean expression ϕ, with Boolean variables partitioned into i sets $X_1, \ldots, X_i$, is it true that for all partial truth assignments for the variables in X_1 there is a partial truth assignment for the variables in X_2 such that for all partial truth assignments for the variables in X_3, and so on up to X_i, ϕ is satisfied by the overall truth assignment? We represent an instance of QSAT$_i$ as follows (by slightly abusing our first-order quantifiers):

$$\exists X_1 \forall X_2 \exists X_3 \ldots Q X_i \quad \phi$$

where, as usual, the quantifier Q is $\exists$ if i is odd and $\forall$ if i is even.

Theorem 17.10: For all $i \geq 1$ QSAT$_i$ is $\Sigma_i\mathbf{P}$-complete.

Proof: Both directions rest heavily on Theorem 17.8 and its Corollary 2. To show that QSAT$_i \in \Sigma_i\mathbf{P}$ we just note that it is defined in the form required by Corollary 2.

To reduce any language $L \in \Sigma_i\mathbf{P}$ to QSAT$_i$, we first bring L in the form of Corollary 2 to Theorem 17.8. Since the relation R can be decided in polynomial time, there is a polynomial-time deterministic Turing machine M that accepts precisely those input strings $x; y_1; \ldots; y_i$ such that $(x, y_1, \ldots, y_i) \in R$. Suppose that i is odd (the even i case is symmetric). Using Cook's theorem (and thus not even taking advantage of the fact that M is deterministic) we can write a Boolean formula ϕ that captures the computation of this machine. The variables in ϕ can be divided into $i + 2$ classes. Variable set X contains the variables standing for the symbols in the input string before the first ";" symbol—recall that the input of M is of the form $x; y_1; \ldots; y_i$. Similarly, variable set Y_1 stands for the next input symbols, and so on up to Y_i. These $i + 1$ sets are called *the input variables*. Finally, there is a (probably much larger) set of Boolean variables Z that incorporates all other aspects of the computation of M.

Now, given any fixed values for the variables in $X, Y_1, \ldots, Y_i$, the resulting expression is satisfiable if and only if the values of the input variables spell a string in the language decided by M, that is, if they are related by R.

Consider now any string x, and substitute in ϕ the corresponding Boolean values $\hat{X}$ for X. We know that $x \in L$ if and only if there is a y_1 such that for all y_2 etc., there is a y_i (remember, i is odd) such that $R(x, y_1, \ldots, y_i)$. But this, in terms of the expression ϕ, means that for these particular values $\hat{X}$ there are values for Y_1 such that for all values of Y_2 etc., there is a value for Y_i *and there is a value of Z*, such that ϕ evaluates to **true**. Thus $x \in L$ if and only if $\exists Y_1 \forall Y_2 \ldots \exists Y_i; Z \phi(\hat{X})$, which is an instance of QSAT$_i$. □

How about the cumulative hierarchy **PH**, does it have complete sets? It turns out that it probably does not. This is not because **PH** is a "semantic class" - it is not. The reason is a little more subtle (compare with Problem 8.4.2).

Theorem 17.11: If there is a **PH**-complete problem, then the polynomial hierarchy collapses to some finite level.

Proof: Suppose that L is **PH**-complete. Since $L \in \mathbf{PH}$, there is an $i \geq 0$ such that $L \in \Sigma_i\mathbf{P}$. But any language $L' \in \Sigma_{i+1}\mathbf{P}$ reduces to L. Since all levels of the polynomial hierarchy are closed under reductions, this means that $L' \in \Sigma_i\mathbf{P}$, and hence $\Sigma_i\mathbf{P} = \Sigma_{i+1}\mathbf{P}$. □

There is a rather obvious upper bound on the power of the polynomial hierarchy: Polynomial space. Indeed, starting fron the characterization in Corollary 2 of Theorem 17.8, it is easy to see that the search for the strings $y_1, y_2, \ldots, y_i$ can comfortably fit within polynomial space. In fact, in the next chapter we shall see that **PSPACE** is in some sense a generalization and extension of the polynomial hierarchy.

Proposition 17.1: $\mathbf{PH} \subseteq \mathbf{PSPACE}$. □

But is $\mathbf{PH} = \mathbf{PSPACE}$? This is an open and intriguing question. However notice this curious fact: If $\mathbf{PH} = \mathbf{PSPACE}$ then by Theorem 17.11 **PH** has complete problems (**PSPACE** has), and thus the polynomial hierarchy collapses. Although the $\mathbf{PH} = \mathbf{PSPACE}$ eventuality would seem to be "stretching" the polynomial hierarchy upwards, and therefore to strengthen it, in fact it does the opposite. Finally, **PH** has a very natural logical characterization (arguably more natural than Fagin's theorem, see Problem 17.3.10).

BPP and Polynomial Circuits

When studying **BPP** in Section 11.2 we noted that it is not known to be contained in **NP** (or **coNP**; since **BPP** is closed under complementation, it is a subset of both or neither). We can now show by a probabilistic technique that it is in the second level of the polynomial hierarchy:

Theorem 17.12: $\mathbf{BPP} \subseteq \Sigma_2\mathbf{P}$.

Proof: Let $L \in \mathbf{BPP}$. All we know about L is that there is a precise Turing machine M, with computations of length $p(n)$ on inputs of length n, that decides L by clear majority. For each input x of length n, let $A(x) \subseteq \{0,1\}^{p(n)}$ denote the set of accepting computations (the choices that lead to "yes.") We can assume that if $x \in L$ then $|A(x)| \geq 2^{p(n)}(1 - \frac{1}{2^n})$, and if $x \notin L$ then $|A(x)| \leq 2^{p(n)}\frac{1}{2^n}$. That is, the probability of a false answer (false positive or false negative) is at most $\frac{1}{2^n}$, instead of the usual $\frac{1}{4}$. This can be assured by repeating the **BPP** algorithm enough times and taking the majority outcome (recall the discussion in Section 11.3).

Let U be the set of all bit strings of length $p(n)$. For $a, b \in U$ define $a \oplus b$ to be the bit string which is the componentwise exclusive or of the two bit strings. For example, $1001001 \oplus 0100101 = 1101100$. This operation has some very useful properties. First, $a \oplus b = c$ if and only if $c \oplus b = a$. That

is, the function "$\oplus b$" applied to a twice gets us back to a. As a result, the function "$\oplus b$" is one-to-one (because its argument can be recovered). Second, if a is a fixed string and r a *random* string, drawn by flipping an unbiased coin independently $p(n)$ times, then $r \oplus a$ is also a random bit string. This is because "$\oplus a$" is a permutation of U, and thus does not affect the uniform distribution.

Let t be a bit string of length $p(n)$, and consider the set $A(x) \oplus t = \{a \oplus t : a \in A(x)\}$. We call this set *the translation of* $A(x)$ *by* t. Since the function $\oplus t$ is one-to-one, the translation of $A(x)$ has the same cardinality as $A(x)$. We shall prove the following intuitive fact: If $x \in L$, since $A(x)$ is so large in this case, we can find a relatively small set of translations that covers all of U. However, if $x \notin L$, then $A(x)$ is so small that no such set of translations can exist.

More formally, suppose that $x \in L$, and consider a *random* sequence of $p(n)$ translations, $t_1 \ldots, t_{p(n)} \in U$; they are obtained by drawing $p(n)^2$ bits independently with probability $\frac{1}{2}$. Fix a string $b \in U$. We say that these translations *cover* b if $b \in A(x) \oplus t_j$ for some $j \leq p(n)$. What is the probability that a point b is covered? $b \in A(x) \oplus t_j$ if and only if $b \oplus t_j \in A(x)$. And since $b \oplus t_j$ is as random as t_j, and we have assumed that $x \in L$, we conclude that $\mathbf{prob}[b \notin A(x) \oplus t_j] = \frac{1}{2^n}$. Therefore, the probability that b is *not* covered by any t_j is precisely that number raised to $p(n)$, $2^{-np(n)}$.

So, every point in U fails to be covered with probability $2^{-np(n)}$; it follows that the probability that there is a point that is not covered is at most $2^{-np(n)}$ times the cardinality of U, or $2^{-(n-1)p(n)} < 1$. Thus, a sequence of randomly drawn translations $T = (t_1, \ldots, t_{p(n)})$ has a positive (in fact, overwhelming) probability that it covers all of U. We must conclude that *there is at least one* T *that covers all of* U.

Conversely, suppose that $x \notin L$. Then the cardinality of $A(x)$ is an exponentially small fraction of that of U, and obviously (for large enough n) there is no sequence T of $p(n)$ translations that cover all of U. We conclude that *there is a sequence* T *of* $p(n)$ *translations that cover* U *if and only if* $x \in L$.

The proof that $L \in \Sigma_2\mathbf{P}$ now follows easily from Corollary 2 of Theorem 17.8: We have shown that L can be written as

$$L = \{x : \text{ there is a } T \in \{0,1\}^{p(n)^2} \text{ such that for all } b \in U \\ \text{there is a } j \leq p(n) \text{ such that } b \oplus t_j \in A(x)\}.$$

But this is precisely the form of languages in $\Sigma_2\mathbf{P}$ according to the corollary. The last existential quantifier "there is a j such that..." does *not* affect the position of L in the polynomial hierarchy: It quantifies over polynomially many possibilities, and is therefore an "or" in disguise. To put it otherwise, the whole line "there is a $j \leq p(n)$ such that $b \oplus t_j \in A(x)$" can be tested in polynomial time by trying all t_j's. □

Since **BPP** is closed under complement, we have in fact proved:

Corollary: $\mathbf{BPP} \subseteq \Sigma_2\mathbf{P} \cap \Pi_2\mathbf{P}$. □

We conclude our discussion of the polynomial hierarchy by an interesting result related to circuit complexity. In Section 14.4 we articulated an important "Conjecture B" that strengthens $\mathbf{P} \neq \mathbf{NP}$, namely that SAT (or any other **NP**-complete problem) has no polynomial circuits (uniform or not). The following result adds much credibility to the conjecture:

Theorem 17.13: If SAT has polynomial circuits, then the polynomial hierarchy collapses to the second level.

Proof: The proof is a nice application of the self-reducibility of SAT (recall the proof of Theorems 13.2 and 14.3). That SAT is self-reducible means that there is a polynomial-time algorithm for SAT that invokes SAT on smaller instances. That is, there is a polynomial-time oracle machine M^{SAT} deciding SAT *with* SAT *as an oracle*, only with the additional restriction that, on input of length n, its oracle string can contain at most $n-1$ symbols.

The proof rests on an important consequence of self-reducibility: *Self-testing.* Suppose that there is a family of polynomial circuits $\mathcal{C} = (C_0, C_1, \ldots)$ deciding SAT. In the proof we shall allow for the self-reducibility machine M^{SAT} to use as its oracle, instead of SAT, *an initial segment* $\mathcal{C}_n = (C_0, C_1, \ldots, C_n)$ *of this family*. That is, once a query appears in its query string, the machine $M^{\mathcal{C}_n}$ invokes not SAT, but the appropriate circuit in the segment, assuming that the length of the query is at most n (and we know that the queries of M have small length). We say that the initial segment $\mathcal{C}_n$ *self-tests* if for all Boolean expressions w of size up to n

$$M^{\mathcal{C}_n}(w) = \mathcal{C}_n(w).$$

That is, all Boolean expressions w fed into the appropriate circuit give the same answer as when they are the input of the self-reducibility machine for SAT, with the circuit segment as oracle. If the self-testing equality holds for all w, this means (by induction of the size of w) that $\mathcal{C}_n$ is indeed a correct initial segment of a circuit family for SAT.

On the assumption that SAT has polynomial circuits we must show that $\Sigma_j\mathbf{P} = \Sigma_2\mathbf{P}$ for all j. By Theorem 17.9 we need only show that $\Sigma_3\mathbf{P} = \Sigma_2\mathbf{P}$. So, we are given an $L \in \Sigma_3\mathbf{P}$ and we have to show it is in $\Sigma_2\mathbf{P}$. We can assume that L is of this form:

$$L = \{x : \exists y \forall z (x, y, z) \in R\},$$

where R is a polynomially balanced relation decidable in **NP**—this is a simple variant of Corollary 2 to Theorem 17.8, with the recursion stopped one step earlier. Since R is decidable in **NP** and SAT is **NP**-complete, there is a reduction

F such that $(x, y, z) \in R$ if and only if the Boolean expression $F(x, y, z)$ is satisfiable. Suppose that, on input x, the largest expression $F(x, y, z)$ that can be constructed is of length at most $p(|x|)$. Since R is polynomially balanced and F polynomial-time, $p(n)$ is a polynomial.

To show that L is in $\Sigma_2\mathbf{P}$, we shall argue that $x \in L$ if and only if the following holds:

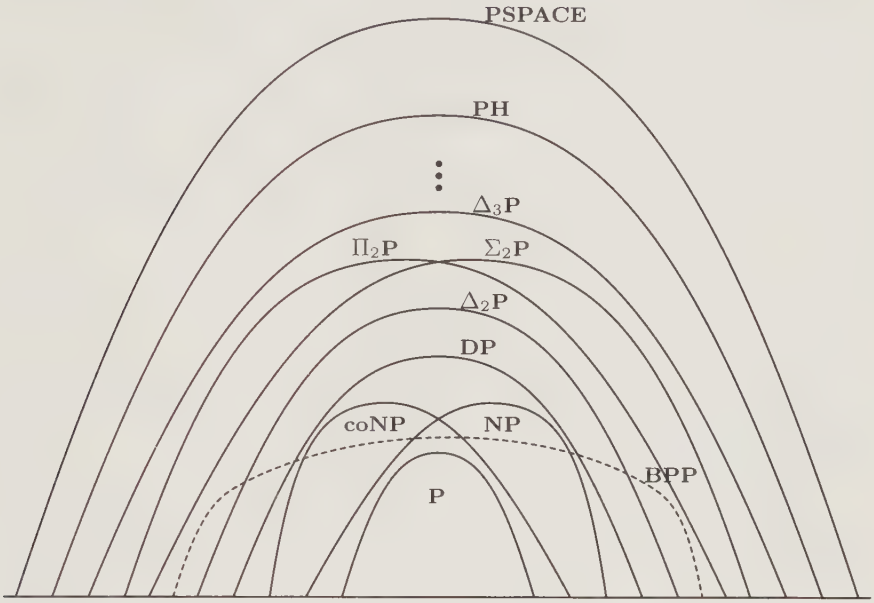
There exists an initial segment $\mathcal{C}_{p(|x|)}$ and there exists a string y such that for all strings z and expressions w –all of length at most $p(|x|)$—we have: (a) $\mathcal{C}_{p(|x|)}$ self-tests successfully on w, that is, $M^{\mathcal{C}_n}(w) = \mathcal{C}_n(w)$, and (b) $\mathcal{C}_{p(|x|)}$ outputs **true** *on expression $F(x, y, z)$.*

Notice that, since the above condition involves two alternations of quantifiers, and the innermost property can be tested in polynomial time, this would settle that $L \in \Sigma_2\mathbf{P}$.

If the above condition holds, then by (a) we know that $\mathcal{C}_{p(|x|)}$ is a correct initial segment of a circuit family for SAT, and thus it can be used to correctly establish in (b) that $R(x, y, z)$, and thus the condition implies $x \in L$. Conversely, if $x \in L$ then there is a y such that for all z $R(x, y, z)$. Furthermore, by our hypothesis that SAT has polynomial circuits, we know that a correct segment exists that will self-test. The same segment will then certify that $(x, y, z) \in R$ for the appropriate y and z. The proof is complete. □

17.3 NOTES, REFERENCES, AND PROBLEMS

17.3.1 Class review:



The class **DP** was introduced in

- C. H. Papadimitriou and M. Yannakakis "The complexity of facets (and some facets of complexity)," *Proc. 24th ACM Symp. on the Theory of Computing*, pp. 229–234, 1982; also, *J.CSS 28*, pp. 244–259, 1984.

Many **DP**-completeness results can be found in this paper, and also in

- C. H. Papadimitriou and D. Wolfe "The complexity of facets resolved," *Proc. 16th IEEE Symp. on the Foundations of Computer Science*, pp. 74–78, 1985; also, *J.CSS 37*, pp. 2–13, 1987.

As for UNIQUE SAT, there is an oracle under which it is *not* **DP**-complete, and so it appears to be a less worthy representative of **DP** than the other problems we have seen:

- A. Blass and Y. Gurevich "On the unique satisfiability problem," *Information and Control, 55*, pp. 80–88, 1982.

But see Problem 18.3.5 in this regard.

The "D" in **DP** stands for "difference": A language in **DP** is just the set-theoretic difference of two languages in **NP**. The corresponding class of differences of two *recursively enumerable languages* was defined in

- H. Rogers *Theory of Recursive Functions and Effective Computability*, MIT Press, Cambridge, Massachusetts, 1987 (second edition).

Incidentally, the class that we call **DP** is denoted in the literature as D^p. We have adopted this new notation, as well as that for the polynomial hierarchy, whose classes are also usually denoted Σ_2^p etc., in order to arrive at a uniform nomenclature for all classes "between" **P** and **PSPACE**: All names end with **P**, and the prefix is indicative of the mode of computation involved.

17.3.2 Problem: (a) Show that the problems CRITICAL SAT, CRITICAL HAMILTON PATH, and CRITICAL 3-COLORABILITY are in **DP**.

(b) Show that UNIQUE SAT is in **DP**.

(c) Show that if UNIQUE SAT is in **NP** then **NP** = **coNP**.

17.3.3 Problem: Show that **DP** $\subseteq$ **PP**.

17.3.4 True or false? (Or equivalent to **P** = **NP**?)
(a) If L is **NP**-complete and L' is **coNP**-complete, then $L \cap L'$ is **DP**-complete.
(b) If L is **NP**-complete $L \cap \overline{L}$ is **DP**-complete.

17.3.5 **DP** can be extended to classes in which an arbitrary *bounded* number of SAT queries are allowed. The resulting *Boolean hierarchy*, somewhat sparse in natural complete problems, was studied in

- J.-Y. Cai, T. Gundermann, J. Hartmanis, L. Hemachandra, V. Sewelson, K. Wagner, and G. Wechsung "The Boolean hierarchy I: Structural properties" *SIAM Journal on Computing 17*, pp. 1232–1252, 1988. Part II: Applications in *vol. 18*, pp. 95–111, 1989.

17.3.6 Show that the following language is Δ_2**P**-complete: Given an instance of the TSP, is the optimum tour length *odd*? Is the optimum tour *unique*?

17.3.7 The relationship between $\mathbf{FP}^{\mathbf{NP}}$ and optimization problems (Theorems 17.5 and 17.6), hinted at in

- C. H. Papadimitriou "The complexity of unique solutions," *Proc. 23rd IEEE Symp. on the Foundations of Computer Science*, pp. , pp. 14–20, 1983; also *J.ACM 31*, pp. 492–500, 1984,

was established in

- M. W. Krentel "The complexity of optimization problems," *Proc. 18th ACM Symp. on the Theory of Computing*, pp. 79–86, 1986; also *J.CSS 36*, pp. 490–509, 1988.

Theorem 17.7 is from

- S. R. Buss and L. Hay "On truth-table reducibility to SAT and the difference hierarchy over NP," *Proc. 3rd Symp. on Structure in Complexity Theory*, pp. 224–233, 1988.

17.3.8 Problem: Show that, if **NP** $\subseteq$ **TIME**$(n^{\log n})$, then **PH** $\subseteq$ **TIME**$(n^{\log^k n})$.

17.3.9 The polynomial hierarchy was introduced and studied in

- L. J. Stockmeyer "The polynomial hierarchy," *Theor. Comp. Science, 3*, pp. 1–22, 1976.

Theorem 17.10 on the completeness of QSAT_i is from

- C. Wrathall "Complete sets for the polynomial hierarchy," *Theor. Comp. Science, 3,* pp. 23–34, 1976.

17.3.10 Show that **PH** is the class of all graph-theoretic properties that can be expressed in *second-order logic.* (Compare with Theorem 8.3.)

17.3.11 Suppose that the cities in a Euclidean instance of the TSP are the vertices of a convex polygon. Then not only is the optimum tour easy to find (it is the perimeter of the polygon), but the instance has the *master tour property:* There is a tour such that the optimum tour of any subset of cities is obtained by simply omitting from the master tour the cities not in the subset.

Problem: Show that deciding whether a given instance of the TSP has the master tour property is in $\Sigma_2\mathbf{P}$.

17.3.12 We know that converting Boolean expressions in disjunctive normal form to conjunctive normal form can be exponential in the worst case, simply because the output may be exponentially long in the input. But suppose the output is small. In particular, consider the following problem: We are given a Boolean expression in disjunctive normal form, and an integer B. We are asked whether the conjunctive normal form has B or fewer clauses.

Problem: Show that the problem is in $\Sigma_2\mathbf{P}$.

Incidentally, the previous two problems are two good candidates for natural $\Sigma_2\mathbf{P}$-complete problems.

17.3.13 Default logic. A *default* is an object of the form $\delta = \frac{\phi:\psi\&\chi}{\chi}$, where ϕ, χ, and ψ are Boolean expressions in conjunctive normal form called the *prerequisite*, the *justification*, and the *consequence* of δ, respectively. Intuitively, the above default means that *if ϕ has been established, and neither $\neg\psi$ nor $\neg\chi$ have been established, then we can "assume χ by default."* For example, here is the intended use of this device in artificial intelligence:

$$\frac{\text{bird(Tweety)} : \neg\ \text{penguin(Tweety)}\&\text{flies(Tweety)}}{\text{flies(Tweety)}}.$$

A *default theory* is a pair $D = (\alpha_0, \Delta)$, where α_0 is a Boolean expression (intuitively, comprising our initial knowledge of the world), and Δ is a set of defaults.

The semantics of a default theory is defined in terms of a peculiar kind of model called an *extension.* Given a default theory (α_0, Δ), an *extension* of (α_0, Δ) is an expression α such that the following sequence of expressions in conjunctive normal form, starting from α_0, converges to α:

$$\alpha_{i+1} = \Theta(\alpha_i \cup \{\chi : \text{for some default } \tfrac{\phi:\psi\&\chi}{\chi} \in \Delta, \\ \alpha_i \Rightarrow \phi \text{ and } \alpha \not\Rightarrow \neg(\psi \wedge \chi)\}).$$

Here $\Theta(\phi)$ denotes the deductive closure, that is, all clauses deducible from ϕ. That is, at each stage we add to α_i all default consequences whose prerequisites have been established already, and whose justifications and consequences do not contradict the

extension sought; we then take all possible logical consequences of the resulting expression. Notice that the sought extension α appears in the iteration. Obviously this process must converge after $|\Delta|$ or fewer steps, but not necessarily to α; if not, α fails to be an extension. Default theories may have one, many, or no extensions. Let DEFAULT SAT be the following problem: "Given a default theory, does it have an extension?"

(a) Show that DEFAULT SAT is $\Sigma_2\mathbf{P}$-complete.

(b) Consider the special case of DEFAULT SAT in which all defaults are of the form $\frac{:x\&y}{x}$, where x and y are literals. Show that DEFAULT SAT in this special case is **NP**-complete.

Default logic was proposed and studied by Ray Reiter

- R. Reiter "A logic for default reasoning," *Artificial Intelligence 13*, 1980.

It is one of the many formalisms representing attempts in artificial intelligence to capture the elusive notion of *common-sense reasoning*, see for example

- M. Genesareth and N. Nielson *Logical Foundations of Artificial Intelligence*, Morgan-Kaufman, San Mateo, California, 1988.

The complexity results in parts (a) and (b) above are from

- C. H. Papadimitriou and M. Sideri "On finding extensions of default theories," *Proc. International Conference in Database Theory*, pp. 276–281, Lecture Notes in Computer Science, Springer-Verlag, 1992.

A very comprehensive complexity-theoretic treatment of this and other formalizations of common-sense reasoning, resulting in several natural problems complete for various levels of the polynomial hierarchy, is contained in

- G. Gottlob "Complexity results in non–monotonic logics," CD–TR 91/24, T. U. Wien, August 1991. Also, *J. of Logic and Computation*, June 1992.

17.3.14 There are now oracles known with respect to which **PH** $\neq$ **PSPACE** and the polynomial hierarchy is infinite, separated from **PSPACE**, or collapses to any desired level, see

- A. C.-C. Yao "Separating the polynomial hierarchy by oracles," *Proc. 26th IEEE Symp. on the Foundations of Computer Science*, pp. 1–10, 1985, also
- J. Håstad *Computational Limitations for Small-depth Circuits*, MIT Press, Cambridge, 1987, and
- K.-I. Ko "Relativized polynomial-time hierarchies with exactly k levels" *SIAM J. Computing, 18*, pp. 392–408, 1989.

Both questions had been open for some time. In fact, separation from **PSPACE** is known to hold for a random oracle

- J.-Y. Cai "With probability one, a random oracle separates PSPACE from the polynomial hierarchy," *Proc. 18th ACM Symp. on the Theory of Computing*, pp. 21–29, 1986; also, *J.CSS, 38*, pp. 68–85, 1988.

17.3.15 A weaker form of Theorem 17.12 was announced in

- M. Sipser "A complexity theoretic approach to randomness," *Proc. 15th ACM Symp. on the Theory of Computing*, pp. 330–335, 1983.

Our proof is from

- C. Lautemann "BPP and the polynomial time hierarchy," *IPL 17*, pp. 215–218, 1983.

Theorem 13.13 is form

- R. M. Karp and R. J. Lipton "Some connections between nonuniform and uniform complexity classes," *Proc. 12th ACM Symp. on the Theory of Computing*, pp. 302 309, 1980; retitled "Turing machines that take advice," *Enseign. Math., 28*, pp. 191–201, 1982,

where its current strong form is attributed to Mike Sipser.

CHAPTER

18 COMPUTATION THAT COUNTS

"...and though the holes were rather small
they had to count them all."

18.1 THE PERMANENT

So far we have studied two related styles of problems: One asks whether a desired solution exists; the other requires that a solution be produced. But there is a third important, natural, and fundamentally different kind of problem: The one that asks *how many solutions exist.*

Example 18.1: Consider the following problem:

#SAT: Given a Boolean expression, compute the number of different truth assignments that satisfy it.

Obviously, if we could solve this problem then we would be able to solve SAT: An expression is satisfiable if and only if this number is non–zero.

Similarly, #HAMILTON PATH asks for the number of different Hamilton paths in the given graph. #CLIQUE the number of cliques of size k or larger. And so on. □

Example 18.2: All problems in the previous example are in effect "counting versions" of **NP**–complete decision problems. But even in cases in which the decision problem is polynomial, counting the solutions may be highly nontrivial. Consider, for example, the MATCHING problem. Although telling whether a bipartite graph has a perfect matching can be done in polynomial time (recall

Section 1.2), computing the number of different perfect matchings in a bipartite graph is an important and notoriously difficult problem.

One could hope to solve the problem of counting perfect matchings by exploiting the connection between matchings and determinants of matrices (recall Sections 11.1 and 15.3). Suppose that $G = (U, V, E)$ is a bipartite graph with $U = \{u_1, \ldots, u_n\}$ and $V = \{v_1, \ldots, v_n\}$, and $E \subseteq U \times V$. Consider the adjacency matrix A^G of the graph, the $n \times n$ matrix whose i, jth element is 1 if $[u_i, v_j] \in E$, and 0 otherwise. The determinant of A^G is

$$\det A^G = \sum_{\pi} \sigma(\pi) \prod_{i=1}^{n} A^G_{i,\pi(i)},$$

where the summation is over all perfect matchings of G. The quantity $\sigma(\pi)$ is -1 if π is the product of an odd number of transpositions, and 1 if it is even. It is this factor that frustrates our plan to count matchings using determinants.

To put it differently, the reason why we can compute determinants efficiently *is precisely this apparent complication of* $\sigma(\pi)$. Because if we get rid of the $\sigma(\pi)$ factor we arrive at another important characteristic of matrices called *the permanent:*

$$\text{perm} A^G = \sum_{\pi} \prod_{i=1}^{n} A^G_{i,\pi(i)}.$$

The permanent of A^G is precisely the number of perfect matchings in G. This is why the problem of counting perfect matchings of bipartite graphs is known not as #MATCHING, but as PERMANENT. We shall see soon that it is a very hard problem.

A bipartite graph G with n "boys" $\{u_1, \ldots, u_n\}$ and n "girls" $\{v_1, \ldots, v_n\}$ can be equivalently thought of as a *directed graph* G' with nodes $\{1, 2, \ldots, n\}$, where we have an edge from i to j in G' if and only if $[u_i, v_j]$ is in G (see Figure 18.1; notice that the directed graph G' may have *self–loops*). It is easy to see that a perfect matching in G corresponds to a *cycle cover of* G'; that is, a set of node disjoint cycles that together cover all nodes of G' (see Figure 18.1). Thus, the permanent of A^G is the total number of all cycle covers of G'. This equivalence provides a better way to visualize matchings and permanents. For example, in Figure 18.1 the bipartite graph has four distinct matchings; naturally enough, the directed graph has four distinct cycle covers, and the permanent of the matrix is 4. □

Example 18.3: Counting solutions is a mode of computation most relevant to probabilistic calculations. For example, REACHABILITY gives rise to the following problem: Given a graph G with m edges, how many of the 2^m subgraphs of G contain a path from node 1 to node n? The question is important because

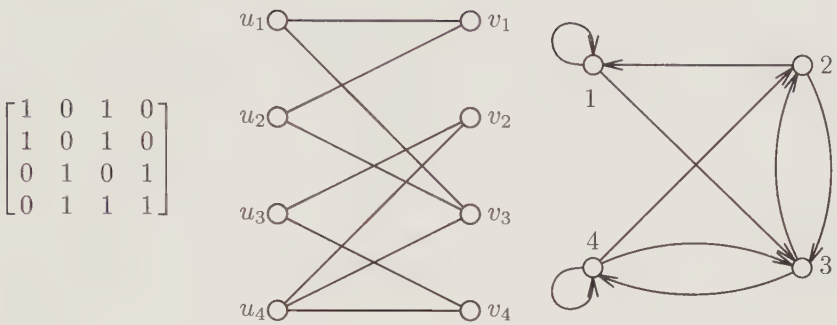


Figure 18-1. Permanents, matchings, and cycle covers.

the portion of subgraphs that connect the two nodes is a precise estimate of the *reliability* of the graph, that is, 2^m times the probability that the two nodes will remain connected if all edges fail independently with probability $\frac{1}{2}$ each. The problem of counting subgraphs that contain a path from 1 to n is therefore called GRAPH RELIABILITY. □

Definition 18.1: We now define a powerful class of functions called **#P** (pronounced "number **P**" or "sharp **P**," or even "pound **P**"). Let Q be a polynomially balanced, polynomial-time decidable binary relation. The *counting problem* associated with Q is the following: Given x, how many y are there such that $(x, y) \in Q$? The output required is an integer in binary, say. **#P** is the class of all counting problems associated with polynomially balanced polynomial-time decidable relations. □

For example, if Q is the relation "y satisfies expression x" then the corresponding counting problem is #SAT. If it is "y is a Hamilton path of graph x" then we have #HAMILTON PATH. By taking Q to be the relation "y is a perfect matching of bipartite graph x" we obtain PERMANENT. If Q is "y is a subgraph of graph x, and there is a path from node 1 to node n in y" we get GRAPH RELIABILITY. And so on. These are important examples of problems in **#P** (soon to be shown **#P**-complete).

As usual with function problems, a reduction between two counting problems A and B consists of two parts: A part R mapping instances x of A to instances $R(x)$ of B, and a part S recovering from the answer N of $R(x)$ the answer $S(N)$ of x. In the case of counting problems there is a convenient kind of reductions, called *parsimonious reductions*. Basically, a reduction is parsimonious if S is the identity function; that is, the number of solutions of instance $R(x)$ is the same as the number of solutions of x. To put it more simply, parsimonious reductions between problems in **FNP** are those that *preserve the*

number of solutions.

The reader may want to look back in Chapter 9 to verify that most reductions between the decision problems in **NP** that we have seen there are indeed parsimonious reductions between the corresponding counting problems (for an exception see Theorem 18.2 below). For example, recall the reduction from CIRCUIT SAT to 3SAT (Example 8.3). It should be clear that there is a one-to-one correspondence between inputs of the circuit that lead to acceptance and satisfying truth assignments of the resulting expression. Therefore the reduction is parsimonious, and the counting problem associated with CIRCUIT SAT reduces to #SAT. We shall use this in the following proof:

Theorem 18.1: #SAT is **#P**-complete.

Proof: This is a parsimonious variant of Cook's theorem. Suppose that we have an arbitrary counting problem in **#P**, defined in terms of the relation Q. We shall show that this problem reduces to #SAT.

We know that Q can be decided by a polynomial-time Turing machine M. We also know that Q is polynomially balanced, that is, for each x the only possible solutions y have length at most $|x|^k$; in fact, it is easy to see that we can assume all solutions have length exactly $|x|^k$, and that the alphabet of the solutions y is $\{0,1\}$. We know from Theorem 8.2 that, based on M and x, we can construct in logarithmic space a circuit $C(x)$, with $|x|^k$ inputs, such that an input y makes the output of $C(x)$ equal to **true** if and only if M accepts $x;y$, or equivalently $(x,y) \in Q$. Thus the construction of $C(x)$ is a parsimonious reduction from the counting problem of Q to the counting problem of CIRCUIT SAT. And the discussion preceding the theorem implies that there is a parsimonious reduction from the counting problem of CIRCUIT SAT to #SAT (parsimonious reductions obviously compose). □

As we have said, most reductions that we have seen in Chapter 9 are, or can be easily made, parsimonious. The reduction from 3SAT to HAMILTON PATH is an exception. It is *not* parsimonious because the completely connected graph of the nodes on the clause side (the marked nodes in Figure 9.7) adds a very large number of Hamilton paths for each satisfying truth assignment (and unfortunately not always the same number). A little more care is needed to prove the following result:

Theorem 18.2: #HAMILTON PATH is **#P**-complete.

Proof: There is a parsimonious reduction from 3SAT to HAMILTON PATH based on the reduction in Theorem 17.5 showing that TSP is $\mathbf{FP}^{\mathbf{NP}}$-complete. Suppose that we modify the graph constructed there (Figure 17.2) as follows: First, we omit the edge between the end of the clause part and the beginning of the variable part (this is the edge e in Figure 17.2), so that we are not seeking a Hamilton cycle, but a Hamilton path between the two top nodes in Figure 17.2. Second, we omit the "emergency edge" of each clause, which is traversed when

the clause is unsatisfied. It is then immediate that the Hamilton paths of the resulting graph are in one-to-one correspondence with the satisfying truth assignments of the given instance of 3SAT. This is because for each satisfying truth assignment there is only one way to traverse the clauses, namely by traversing the edge that corresponds to the *first* **true** literal of the clause. □

The most impressive and interesting #**P**-complete problems *are those for which the corresponding search problem can be solved in polynomial time.* The PERMANENT problem for 0-1 matrices, which is equivalent to the problem of counting perfect matchings in a bipartite graph (or cycle covers in a directed graph, recall Figure 18.1) is the classical example here:

Theorem 18.3 (Valiant's Theorem): PERMANENT is #**P**-complete.

Proof: We shall reduce #SAT to PERMANENT. We are given a set of clauses with three literals each; we must construct a directed graph G such that the cycle covers of G somehow correspond to the satisfying truth assignments of the expression (but, since telling whether a graph has a cycle cover is easy, the correspondence cannot be very direct).

Our construction is very much in the style of completeness proofs for problems of the "Hamilton" type (after all, a Hamilton cycle is certainly a cycle cover). For example, for each variable the graph will have a copy of the *choice gadget*, the directed graph shown in Figure 18.2, where no nodes are shared with other parts of the graph (but there will be communication with other parts of the graph via exclusive-or gadgets attached to the edges). In any cycle cover the two nodes of the graph must be covered by either the cycle to the left (which corresponds to $x =$ **true**) or the cycle to the right ($x =$ **false**).

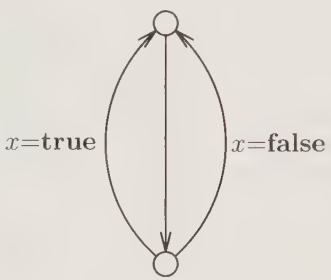


Figure 18-2. The choice gadget for PERMANENT.

For each clause we have a copy of the graph shown in Figure 18.3. The three "external" edges are all connected by exclusive-or's with the edges of the choice gadgets that correspond to the three literals (exactly as we did for HAMILTON PATH). The clause gadget has the following crucial property, easily checked with a little experimentation: *There is no cycle cover in the graph in*

which all three external edges are traversed. And for any proper subset of the external edges (including the empty set), there is exactly one cycle cover that traverses the external edges in the set and no other external edges. Notice what this means: There is a cycle cover of the clause gadget if and only if the truth assignment chosen at the choice gadgets satisfies the clause. It appears that we are very close to the end of a proof (and the correspondence between cycle covers and satisfying truth assignments is surprisingly direct, recall the remark in the first paragraph of the proof).

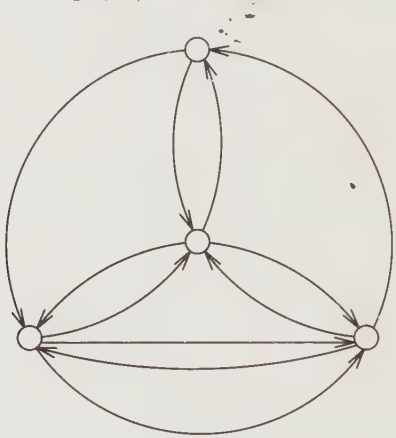


Figure 18-3. The clause gadget.

The catch is, of course, that the exclusive-or gadget has to be remarkably complicated and indirect. We show the exclusive-or gadget in Figure 18.4 (in Figure 18.4(a) as a matrix, in 18.4(b) as a directed graph). Notice immediately that this gadget is a departure from our PERMANENT problem, in that the matrix has entries other than 0 and 1; we will see how to get rid of such entries later. The corresponding graph is also shown (some edges have weights other than one). The *weight* of a cycle cover is now the product of the weights of all its edges (possibly a negative number). This generalization of the PERMANENT problem to matrices with general entries entails computing the sum of the weights of all cycle covers.

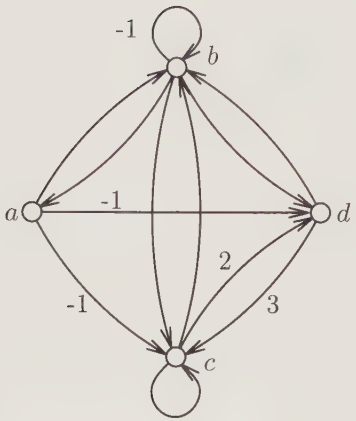
We first state the crucial property of the exclusive-or gadget in terms of matrices and permanents: The matrix of Figure 18.4(a) has the following remarkable properties, easily checkable by a calculation:

(a) The permanent of the whole matrix is 0.

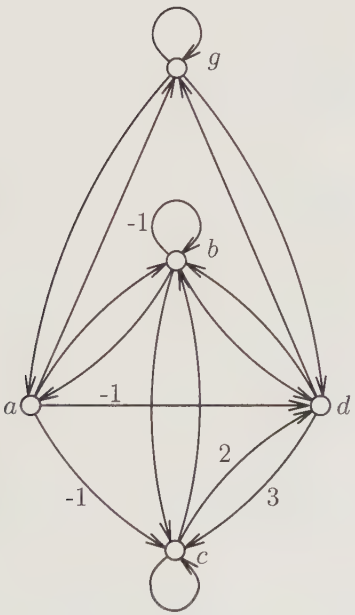
(b) The permanent of the matrix resulting if we delete the first row and column is 0. Similarly for the last row and column. Similarly for the matrix that results if we delete both the first and the last rows and columns.

$$\begin{bmatrix} 0 & 1 & -1 & -1 \\ 1 & -1 & 1 & 1 \\ 0 & 1 & 1 & 2 \\ 0 & 1 & 3 & 0 \end{bmatrix}$$

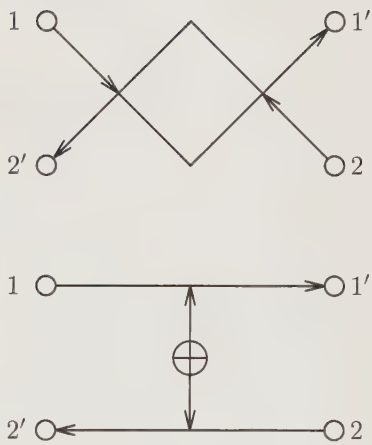
(a)



(b)



(c)



(d)

Figure 18-4. The exclusive-or gadget.

(c) The permanent of the matrix resulting if we delete the first row and the last column is 4. Similarly for the last row and first column.

We can restate this in terms of the graph in Figure 18.4(c) (intuitively, the added node g stands for the rest of the graph). The total weight of all cycle covers of this graph is 8. Of these, a weight of 4 comes from the cycle covers contain the edges (g, d) and (a, g) (this corresponds to deleting the first row and last column of the matrix), and another 4 from cycle covers containing the edges (g, a) and (d, g) (this corresponds to deleting the first column and the last row). The contribution of all cycle covers that contain the loop on g (no deleted rows and columns) is zero. Similarly, the contribution of all cycle covers where the edges (g, a) and (a, g) are traversed (delete the first row and column) is zero, and so is the contribution of all cycle covers containing the edges (g, d) and (d, g) (delete the last row and column). Finally, the contributions of all cycle covers containing the loop on g is also zero.

In terms of the overall graph, this property states that the graph in Figure 18.4(b) (abbreviated as in Figure 18.4(d)) behaves as a valuable exclusive-or gadget. Suppose that we have a graph G containing two edges $(1, 1')$ and $(2, 2')$. Now connect these two edges by the four-node graph, as in Figure 18.4(d). It follows from the properties in the previous paragraph that the total sum of the cycle covers of G in which edge $(1, 1')$ is traversed, but edge $(2, 2')$ is not, is multiplied by 4; similarly for those that traverse edge $(2, 2')$ but not edge $(1, 1')$. But all other cycle covers of G will now contribute zero to the final count, and are thus effectively forbidden.

Our construction is now complete: We connect each external edge of each clause gadget with the choice edge that corresponds to its literal by an exclusive-or. We claim that *the total weight of all cycle covers of the resulting graph is $4^m s$, where m is the total number of literal occurrences in the given expression (the number of exclusive-or gadgets in our graph) and s is the number of satisfying truth assignments* of the given expression. The proof follows rather easily from the above discussion: Any cycle cover that does not comply with all exclusive-or gadgets contributes zero to the total. Any other cycle cover must correspond to a satisfying truth assignment (since it traverses all clause gadgets) and contributes 4^m (since it also traverses all of the m exclusive-or gadgets, with each contributing a factor of four). We conclude that we #SAT can be reduced to the generalization of PERMANENT to the domain of integer, not zero-one, matrices.

To complete the proof, we must now show how we can simulate integer entries by $0 - 1$ entries, so that the permanent is preserved. For small positive entries such as 2 and 3 in Figure 18.4(a) this is easy to do: Just replace the edge with weight 2, say, with the gadget shown in Figure 18.5(a); obviously, this gadget contributes a weight of two to any cycle cover that contains the edge it replaces. Similarly for three (Figure 18.5(b)). Even if we had a large integer entry, such as 2^n, we could simulate it by arranging n such gadgets *in tandem* (Figure 18.5(c)). The only difficulty that remains is the -1 entries.

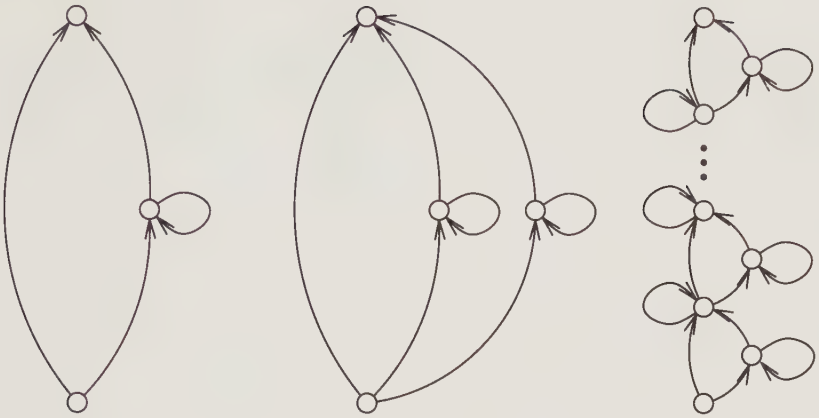


Figure 18-5. Simulating positive weights.

To deal with the -1 entries we must consider another problem closely related to PERMANENT, namely PERMANENT MOD N: We are given a $0-1$ matrix A and an integer N; we are asked to find the value perm$A$ modulo N. This problem as introduced is not necessarily in **#P**, because there is no obvious search problem for which it is the corresponding counting problem. However, it is clear that PERMANENT MOD N reduces to PERMANENT (if we could calculate the permanent exactly, then we could surely take its residue modulo N). Therefore, to complete our proof that PERMANENT is **#P**-complete, it suffices to reduce #SAT to PERMANENT MOD N, instead of PERMANENT.

The trick is that, now that we calculate the permanent of the matrix under construction modulo an integer N, *we can consider the bothersome* -1*'s as* $N-1$*'s*. Let us choose $N = 2^n + 1$, where n is a suitably large integer (say, $n = 8m$) so that the permanent of the matrix we have constructed so far does not exceed N. We can now replace the -1's by 2^n, powers of two that can be simulated as in Figure 18.5(c). The permanent of the resulting matrix modulo N is precisely 4^m times the number of satisfying truth assignments of the original formula (the "modulo N" part of the statement is where the correspondence between satisfying truth assignments and cycle covers finally breaks down). The proof is complete. □

18.2 THE CLASS ⊕P

It is easy to see that any counting problem in **#P** can be solved in polynomial space: reusing space we can enumerate all solutions in lexicographic order, keeping a counter of the ones that we have seen. Hence **#P**, like the polynomial

hierarchy of the previous chapter, is no more powerful than polynomial space. The question is, how do these important generalizations of **NP**—the polynomial hierarchy and **#P**—compare in power? Intuitively, PERMANENT and the other **#P**-complete problems appear to be extremely hard, even in the company of the formidable problems in the polynomial hierarchy. It seems plausible to conjecture that **#P** is more powerful than the polynomial hierarchy, that counting takes you further than quantifiers.

For once, *this conjecture can actually be proved.* Since we cannot compare immediately **#P**, a class of functions, to **PH**, a class of languages, to state the result precisely we recall the class **PP** (Chapter 11) of problems asking whether more than half of the computations of a nondeterministic machine are accepting. This class is closely related to **#P**: Problems in it can be thought of as asking whether *the first bit* of the number of accepting computations (surely a number between 0 and 2^n where n is the number of steps in the computation) is zero or one—whereas **#P** asks for all n bits of this number. *Toda's theorem* (see the references in 18.3.4) states that

$$\mathbf{PH} \subseteq \mathbf{P}^{\mathbf{PP}} \tag{1}$$

That is, polynomial-time oracle machines with a **PP** oracle can decide all languages in the polynomial hierarchy. Counting is indeed very powerful.

In this section we show a simpler result in the same direction of demonstrating the power of counting. We consider problems that require not the first bit of the number of solutions, *but the last.* For example, consider the following two problems:

⊕SAT: Given a set of clauses, is the number of satisfying truth assignments odd?

⊕HAMILTON PATH: Given a graph, does it have an odd number of Hamilton paths? (For this problem to make sense we must consider two Hamilton paths that use the same edges in opposite directions as the same path.)

In general, we say that a language L is in the class $\oplus\mathbf{P}$ (pronounced "odd **P**" or "parity **P**") if there is a nondeterministic Turing machine M such that for all strings x we have $x \in L$ if and only if the number of accepting computations of M on input x is odd. Equivalently, if there is a polynomially balanced and polynomially decidable relation R such that $x \in L$ if and only if the number of y's such that $(x, y) \in R$ is odd. The following two results concerning $\oplus\mathbf{P}$ are immediate:

Theorem 18.4: ⊕SAT and ⊕HAMILTON PATH are $\oplus\mathbf{P}$-complete.

Proof: That they are in $\oplus\mathbf{P}$ follows easiest from the second definition of $\oplus\mathbf{P}$ above. Completeness follows from the parsimonious reductions of any problem in **#P** to #SAT, and from that to #HAMILTON PATH. □

Theorem 18.5: $\oplus\mathbf{P}$ is closed under complement.

Proof: The complement of $\oplus$SAT (telling whether there is an *even* number of satisfying truth assignments) is obviously $\mathbf{co}\oplus\mathbf{P}$-complete. Now this language reduces to $\oplus$SAT, as follows: Given any set of clauses on n variables $x_1, \ldots, x_n$, add the new variable z, add to all clauses the literal z, and add the n clauses $(z \Rightarrow x_i)$ for $i = 1, \ldots, n$. Any satisfying truth assignment of the old expression is still satisfying (with $z =$ **false**), and we have the extra all-**true** satisfying truth assignment (the only one with $z =$ **true**). Hence the number of satisfying truth assignments has been increased by one, completing the reduction from the complement of $\oplus$SAT to $\oplus$SAT. Since $\oplus$SAT is both $\oplus\mathbf{P}$-complete and $\mathbf{co}\oplus\mathbf{P}$-complete, and these classes are closed under reductions (easy to check), it follows that $\oplus\mathbf{P} = \mathbf{co}\oplus\mathbf{P}$. □

In contrast to **PP**, which appears to encapsulate all of the strength of **#P**, $\oplus\mathbf{P}$ captures a fairly weak and benign aspect of counting: The parity of the number of solutions. As partial evidence of how weak this is, notice that PERMANENT MOD 2—*alias* $\oplus$MATCHING, the problem of telling whether the number of perfect matchings of a given bipartite graph is odd—is in **P** (the determinant of the matrix M^G modulo 2 now provides the correct answer). Still, we can show that, if an **RP** machine is equipped with an $\oplus\mathbf{P}$ oracle, it can simulate all of **NP** (Theorem 18.6 below). Compared with Toda's theorem (equation (1) above), this result uses oracle machines that are more powerful (but still "arguably practical" when used without oracles), and a much weaker oracle; the class captured is the lowest level of **PH**. Interestingly, the technique in the proof of Theorem 18.6 is one of the many ingredients of the proof of Toda's theorem.

Theorem 18.6: $\mathbf{NP} \subseteq \mathbf{RP}^{\oplus\mathbf{P}}$.

Proof: We shall describe a polynomial Monte Carlo algorithm for SAT using an oracle for $\oplus$SAT. We need some preliminary definitions: Suppose that we are dealing with a Boolean expression ϕ in conjunctive normal form with n Boolean variables $x_1, \ldots, x_n$. Let $S \subseteq \{x_1, \ldots, x_n\}$ be a subset of the variables. The *hyperplane* η_S is the Boolean expression stating that *an even number among the variables in* S *are* **true**. Let $y_0, \ldots, y_n$ be new variables. η_S can be expressed as the conjunction of the clauses (y_0), (y_n), plus, for $i = 1, \ldots, n$, the following expression: $(y_i \Leftrightarrow (y_{i-1} \oplus x_i))$ if $i \in S$, and $(y_i \Leftrightarrow y_{i-1})$ if $i \notin S$. Naturally, each of these expressions, involving at most three variables, can be easily rewritten in conjunctive normal form. Intuitively, adding the clauses of the hyperplane η_S to an expression ϕ has the effect of intersecting the set of satisfying truth assignments of ϕ by a hyperplane in the n-dimensional vector space modulo 2. The point is that, if we continue intersecting our expression with random hyperplanes n times, then with reasonable probability one of the resulting expressions has a single satisfying truth assignment (and thus its satisfiability can

be detected by our $\oplus$SAT oracle).

Our Monte Carlo algorithm for SAT, with oracle $\oplus$SAT, is this:

> Let ϕ_0 be the given expression ϕ. For $i = 1, \ldots, n+1$ repeat the following:
> Generate a random subset S_i of the variables, and set $\phi_i = \phi_{i-1} \wedge \eta_{S_i}$.
> If $\phi_i \in \oplus$SAT, then answer "ϕ is satisfiable."
> If after $n+1$ steps none of the ϕ_i's is in $\oplus$SAT,
> then answer "ϕ is probably unsatisfiable."

Obviously, the algorithm has no false positives: If the number of satisfying truth assignments of ϕ_i is odd then it is certainly non-zero. So, ϕ_i is satisfiable, and since ϕ_i is ϕ with some additional clauses, ϕ is satisfiable as well. But the algorithm can have false negatives: An expression may have, say, two satisfying truth assignments, and both of them could be eliminated by the first hyperplane chosen. *We shall prove that the probability of a false negative is no larger than* $\frac{7}{8}$ (by repeating the algorithm six times the probability of a false negative becomes less than half, as required by our definition of **RP**).

That false negatives have probability at most $\frac{7}{8}$ relies on the following claim:

> **Claim:** If the number of satisfying truth assignments of ϕ is between 2^k and 2^{k+1} where $0 \leq k < n$, then the probability that ϕ_{k+2} has exactly one satisfying truth assignment is at least $\frac{1}{8}$.
>
> ***Proof of the claim:*** Let T be the set of satisfying truth assignments of ϕ; we are assuming that $2^k \leq |T| \leq 2^{k+1}$. Let us say that two truth assignments *agree* on η_S if either they both satisfy η_S or they both falsify it. Let us now fix $t \in T$, and consider another element $t' \in T$. The probability that t' agrees with t on all $k+2$ first hyperplanes is $\frac{1}{2^{k+2}}$ (since this would mean that all $k+2$ first S_i's contain an even number of variables where t and t' disagree, and these events are independent with probability $\frac{1}{2}$ each). Summing over all $t' \in T - \{t\}$ we see that the probability that t agrees with some t' on the first $k+2$ hyperplanes is at most $\frac{|T|-1}{2^{k+2}} < \frac{1}{2}$. We conclude that the probability that t disagrees with every other member of T on one of the first $k+2$ hyperplanes is at least $\frac{1}{2}$.
>
> Now, the probability that t satisfies all $k+2$ first hyperplanes is obviously $\frac{1}{2^{k+2}}$; and we saw in the previous paragraph that, if it does satisfy them, then with probability at least $\frac{1}{2}$ it is the only one that does (it is easy to see that the fact that t satisfies the $k+2$ first hyperplanes does not affect the probability that it disagrees with the rest of T). So, *with probability at least* $\frac{1}{2^{k+3}}$ t *is the unique satisfying truth assignment of* ϕ_{k+2}. Since this holds for each element t of T,

and T has at least 2^k elements, the probability that such an element of T exists is at least $2^k \times \frac{1}{2^{k+3}} = \frac{1}{8}$. □

If the number of satisfying truth assignments of ϕ is not zero, then it is between 2^k and 2^{k+1} for *some* $k < n$. It follows that at least one of the ϕ_i's will have probability at least $\frac{1}{8}$ to be satisfied by a unique truth assignment—and thus by an odd number of truth assignments. The theorem has been proved. □

18.3 NOTES, REFERENCES, AND PROBLEMS

18.3.1 The verse in the chapter header is from the song "A Day in the Life" by The Beatles, an English rock group in the 1960s.

18.3.2 $\#\mathbf{P}$ was introduced in

- L. G. Valiant "The complexity of computing the permanent," *Theoretical Comp. Science, 8*, pp. 189–201, 1979,

where Theorem 18.3 is proved—in fact, the slightly weaker version stating that $\#\mathbf{P} \subseteq \mathbf{P}^{\text{PERMANENT}}$. That PERMANENT is $\#\mathbf{P}$-complete under reductions is pointed out in

- V. Zankó "$\#\mathbf{P}$-completeness via many-one reductions," *Intern. J. Foundations of Comp. Science, 2*, pp. 77–82, 1991.

More $\#\mathbf{P}$-completeness results were shown in

- L. G. Valiant "The complexity of enumeration and reliability problems," *SIAM J. Computing 8*, pp. 410–421, 1979,
- M. E. Dyer, A. M. Frieze "On the complexity of computing the volume of a polyhedron," *SIAM J. Computing 18*, pp. 205–226, 1989.

18.3.3 Problem: Show that $\mathbf{P}^{\mathbf{PP}} = \mathbf{P}^{\#\mathbf{P}}$. That is, polynomial algorithms with majority oracles are as powerful as those with exact count oracles. Note that we have to first modify oracle machines so that they receive output from their queries; alternatively we could define $\mathbf{P}^{\#\mathbf{P}}$ to be the class of languages decided in polynomial time by oracle machines with queries of the form "is the permanent of matrix A at most K?" (This result is from

- D.Angluin "On counting problems and the polynomial hierarchy," *Theoretical Computer Science, 12*, pp. 161–173, 1980.)

18.3.4 Toda's theorem is from

- S. Toda "On the computational power of PP and $\oplus$P," *Proc. 30th IEEE Symp. on the Foundations of Computer Science*, pp. 514–519, 1989.

For a different proof see

- L. Babai and L. Fortnow "A characterization of #P by arithmetic straight-line programs," *Proc. 31st IEEE Symp. on the Foundations of Computer Science*, pp. 26–35, 1990.

The class $\oplus\mathbf{P}$ was introduced in

- C. H. Papadimitriou, S. Zachos "Two remarks on the power of counting," *Proc. 6th GI Conference in Theoretical Computer Science*, Lecture Notes in Computer Science, Volume 145, Springer Verlag, Berlin, pp. 269–276, 1983.

Theorem 18.6 is from

- L. G. Valiant, V. V. Vazirani "NP is as easy as detecting unique solutions," *Theor. Comp. Science, 47*, pp. 85–93, 1986.

18.3.5 Problem: (a) Suppose that S is a set of nonnegative integers such that $1 \in S$ and $0 \notin S$. Suppose that **SP** is the class of all languages decided by nondeterministic Turing machines with the following acceptance convention: Input x is accepted if the number of accepting computations is in S. Generalize Theorem 18.6 to show that $\mathbf{NP} \subseteq \mathbf{RP}^{\mathbf{SP}}$.

(b) Show that $\mathbf{DP} \subseteq \mathbf{RP}^{\text{UNIQUESAT}}$ (recall Section 17.1). That is, UNIQUE SAT is **DP**-complete after all (recall the references in Chapter 17), albeit under randomized reductions. (Both results are from the paper by Valiant and Vazirani cited above.)

18.3.6 Problem: Show that $\mathbf{UP} \subseteq \oplus\mathbf{P}$ (recall Section 12.1).

For a nontrivial extension of this result to a generalization of **UP** see

- J.-Y. Cai, L. Hemachandra "On the power of parity polynomial time," pp. 229–239 in *Proc. 6th Annual Symp. on Theor. Aspects of Computing*, Lecture Notes in Computer Science, Volume 349, Springer Verlag, Berlin, 1989.

CHAPTER

19 POLYNOMIAL SPACE

Polynomial space-bounded computation has a surprising variety of alternative characterizations. It also has many and diverse kinds of natural complete problems—which is another way of saying the same thing.

19.1 ALTERNATION AND GAMES

Perhaps the most fundamental complete problem for **PSPACE** is *quantified satisfiability*, or QSAT[†]: Given a Boolean expression ϕ in conjunctive normal form, with Boolean variables $x_1, \ldots, x_n$, is it true that for both truth values for the variable x_1 there is a truth value for the variable x_2 such that for both truth values for the variable x_3, and so on up to x_n (where the nth quantifier is "for all" if n is even, and "exists" if n is odd), ϕ is satisfied by the overall truth assignment? In other words,

$$\exists x_1 \forall x_2 \exists x_3 \ldots Q_n x_n \quad \phi?$$

Recall that we have already seen a variant of this problem: In the $\Sigma_i\mathbf{P}$-complete problem QSAT_i we had an *a priori* bound i on the number of alternations of quantifiers allowed (recall Section 17.2). Notice that QSAT is indeed a generalization of all the QSAT_i's, despite the apparent restriction of strictly alternating quantifiers (whereas QSAT_i allows consecutive similarly quantified variables): To ensure strict alternation we may insert to the prefix appropriately quantified "dummy" variables that do not appear in ϕ.

---

† This problem is usually known as QBF, for *quantified Boolean formula;* we use QSAT to emphasize that it is yet another version of satisfiability, capturing yet another important level of complexity.

Theorem 19.1: QSAT is **PSPACE**-complete.

Proof: To show that QSAT can be solved in polynomial space, suppose that we are given a quantified Boolean expression

$$\exists x_1 \forall x_2 \exists x_3 \ldots Q_n x_n \quad \phi.$$

All possible truth assignments of the variables can be arranged as the leaves of a full binary tree of depth n. The left subtree of the root contains all truth assignments with $x_1 =$ **true**, and the right subtree those with $x_1 =$ **false**; then we branch on x_2, then x_3, and so on. We can turn this tree into a *Boolean circuit* , where all gates at the ith level are AND gates if i is even (that is, if x_i is universally quantified), and we have OR gates at the odd levels. An input gate (a leaf of the tree, that is) is **true** if the corresponding truth assignment satisfies ϕ, and it is **false** otherwise. It is immediate from this construction that the given quantified expression is a "yes" instance of QSAT if and only if the value of this circuit is **true**. We can now evaluate this circuit in $\mathcal{O}(n)$ space using the technique we developed in the proof of Theorem 16.1 for computing the value of circuits in space proportional to their depth.

But of course there is a familiar difficulty: The binary tree and circuit used in this algorithm are of exponential size, and thus we cannot afford to store them. But we know from the proof of Proposition 8.2 that space-bounded algorithms (be it logarithmic-space or at any higher level) can be combined.

We must now show that all problems in **PSPACE** reduce to QSAT. The proof uses the reachability method for dealing with space-bounded computations (recall Section 7.3); in fact, the proof is essentially a restatement of the proof of Savitch's Theorem (Theorem 7.5) in the language of logic.

Suppose that L is a language decidable by a Turing machine M which, on input x uses polynomial space. To decide whether $x \in L$, where $|x| = n$, we consider the configuration graph of M on input x. We know that it has at most 2^{n^k} configurations for some integer k. We can thus encode a configuration of M on input x as a bit vector of length n^k.

Recall that in the proof of Savitch's algorithm we determine whether $x \in L$ by computing a Boolean function PATH(a, b, i), which is **true** if and only if there is a path from a to b of length at most 2^i, where a and b are configurations and i is an integer. In the present proof we shall show how, for each integer i, to write a quantified Boolean expression ψ_i with free Boolean variables (i.e., variables not bound by any quantifier) in the set $A \cup B = \{a_1, \ldots, a_{n^k}, b_1, \ldots, b_{n^k}\}$, such that ψ_i is **true** for some truth assignment to its free variables if and only if the truth assignment for the a_i's and the b_i's encodes two configurations a and b such that there is a path from a to b in the configuration graph of length at most 2^i. Once we show how to carry out this construction, $x \in L$ would be expressed as $\psi_{n^k}(A, B)$, where we have substituted for A the truth assignment

that encodes the initial configuration, and for B the accepting configuration (let us assume without loss of generality that it is unique).

For $i = 0$, $\psi_0(A, B)$ simply states that either $a_i = b_i$ for all i, or the configuration B follows from A in one step. It is easy to see that ψ_0 can be written the disjunction of $\mathcal{O}(n^k)$ implicants, each containing $\mathcal{O}(n^k)$ literals (why we prefer the disjunctive normal form will be clear later in the proof).

Inductively, suppose that we have $\psi_i(A, B)$. It is tempting to define $\psi_{i+1}(A, B)$ simply as

$$\exists Z[\psi_i(A, Z) \wedge \psi_i(Z, B)],$$

where Z is a new block of variables encoding the midpoint of the path. Unfortunately, this construction would produce *exponentially large expressions* (since the length of the expression at least doubles in size in each iteration); as a consequence, this would not be a logarithmic-space reduction.

The clever way to circumvent this difficulty is to *use the same copy of ψ_i for asserting existence of paths from a to z and from z to b*. This can be done by writing $\psi_{i+1}(A, B)$ as follows:

$$\exists Z \forall X \forall Y \left[((X = A \wedge Y = Z) \vee (X = Z \wedge Y = B)) \Rightarrow \psi_i(X, Y)\right],$$

where X, Y, and Z are blocks of n^k variables. That is, we want $\psi_i(X, Y)$ to hold whenever either $X = A$ and $Y = Z$, or $X = Z$ and $Y = B$. Notice that "reusing the expression $\psi_i(X, Y)$" is the logical equivalent of "reusing space" for the two recursive calls of PATH in the proof of Theorem 7.5.

As constructed, ψ_{i+1} is not in the form required by QSAT. First, it is not in prenex form, with all quantifiers in front, since the quantifiers of ψ_i are separated from the "new" quantifiers $\exists Z \forall X \forall Y$. This is easy to fix: By the properties of quantifiers (2) and (3) in Proposition 5.10, the quantifiers of ψ_i can migrate in the front, so that they come immediately after the new ones.

A slightly more serious problem is that the matrix of ψ_{i+1} is not in conjunctive normal form, as required by the definition of QSAT. It *seems to be* a serious problem, since the conjunctive normal form of ψ_{i+1}, even treating ψ_i as a single variable, can be shown to require exponentially many clauses. Fortunately, the *disjunctive* normal form of ψ_{i+1} is small and easy to compute. It consists of the disjunctive normal form of ψ_i, followed by $16n^{2k}$ implicants. For each choice of two integers i, j between 1 and n^k, there are sixteen implicants in the list. The first one is

$$(x_i \wedge \neg a_i \wedge x_j \wedge \neg z_j),$$

capturing one way in which $(X = A \wedge Y = Z) \vee (X = Z \wedge Y = B)$ can be made **false**. The other fifteen vary on whether the first or the second literal in each pair is required to be **true** (four combinations), and whether they deal with the

first or the second conjuct in $(X = A \wedge Y = Z)$ and $(X = Z \wedge Y = B)$ (four combinations). Notice that the construction of ψ_{n^k} can indeed be performed in logarithmic space: We add to ψ_0 n^k layers of new sets of clauses (the negations of the implicants discussed above), each on a different block of variables, and we finally prefix the result with n^k layers of quantifiers.

We have thus described a reduction from any problem in **PSPACE** to the version of QSAT in which the matrix is in disjunctive, not conjunctive, normal form. But this version of QSAT is precisely *the complement* of QSAT. Thus, every language in **PSPACE** can be reduced to the complement of QSAT. It follows that any problem in **coPSPACE** can be reduced to QSAT; but of course **PSPACE** = **coPSPACE**, and the proof is complete. □

Recall now the class **AP** = **ATIME**(n^k) of languages decided in polynomial time by *alternating* Turing machines (Section 16.2). It should be no surprise that QSAT is complete for this class:

Theorem 19.2: QSAT is **AP**-complete.

Proof: That QSAT can be solved in alternating polynomial time is immediate: The computation will guess the truth values of the variables $x_1, x_2, \ldots$ one-by-one, where existentially quantified variables are guessed at states in K_{OR}, while universally quantified ones at states in K_{AND}. A final state is accepting if the guessed truth assignment satisfies the expression, and rejecting otherwise. It follows from the definition of acceptance for alternating machines that a quantified expression is accepted if and only if it is **true**; the time needed is polynomial.

The proof of completeness is a variant of Cook's Theorem (Theorem 8.2). As with a nondeterministic machine, the computation of a polynomial-time alternating Turing machine on a given input can be captured by a table, with extra nondeterministic choices. The only difference is that now in the resulting expression the quantifiers for the nondeterministic choices are universal if the current state is in K_{AND}, and existential if in K_{OR}. We can standardize our alternating machines so that the successors of K_{OR} configurations are K_{AND}, and vice-versa; so the variables standing for nondeterministic choices at even levels are existentially quantified, and at odd levels universally. All other variables (the gates of the circuit) are quantified existentially. It is easy to see that the machine accepts the input if and only if the resulting quantified expression is **true**. And of course we can make the quantified expression strictly alternating by adding dummy variables. □

Corollary: AP = PSPACE.

Proof: The two classes are closed under reductions, and QSAT is complete for both. □

QSAT is our first specimen of an interesting genre of **PSPACE**-complete

problems: *Two-person games.* QSAT can be considered as a game between two players, call them $\exists$ and $\forall$. The two players move alternatingly, with $\exists$ moving first. A move consists of determining the truth value of the next variable—in the ith move, if i is odd then $\exists$ fixes the value of x_i, whereas if i is even $\forall$ fixes the value of x_i. $\exists$ tries to make the expression ϕ **true**, while $\forall$ tries to make it **false**. Clearly, after n moves (where n is the number of variables) one of the two players is going to win. By a *game* in this chapter we shall mean a situation such as QSAT: Two players alternate moving, where each move consists of chosing among several predetermined possibilities for changing the "board"—in our case, the expression and a partial truth assignment. The number of moves is bounded by a polynomial in the size of the board. In the end, certain board positions are considered a win for one player (those that satisfy ϕ are a win for $\exists$), while the rest are a win for the other (and it is easy to see which is which). Many common board games (such as chess, checkers, Go, nim, tic-tac-toe, and so on, see the rest of this section and the references) are of this form.

Notice that in a two-person game such as this, a "solution" is not a simple succinct object like a satisfying truth assignment or a cheap tour. What is required is a full-fledged *strategy* for one player, that is, a successful response to any position and move by the other player. Such a strategy is in general an object of exponential size.

It would be very interesting to be able to use **PSPACE**-completeness in order to separate those board games that are intuitively hard (like chess, checkers and Go) from those that are easy (such as tic-tac-toe and nim). Unfortunately, there is a problem: Board games are typically defined on a board of some fixed size, and in fact the size of the board is an important part of the definition of the game. All finite games can in principle be played optimally and fast by a Turing machine that has encoded in its huge set of states all possible positions and moves! Computational complexity as developed in this book, ignoring the "descriptional" complexity of the machines involved, does not seem to be appropriate for exploring the differences in the difficulty of finite board games.

We can generalize many finite board games so that they are played on an arbitrary $n \times n$ board. Some games, like chess, do not have a natural generalization, since the size of the board and the six ranks of chess pieces seem to be indispensable ingredients of its definition, and the essence of chess would be hopelessly distorted if these parameters were altered (but see the references for complexity results concerning such generalizations of chess). On the other hand, games like checkers, Go, nim, tic-tac-toe, etc. seem to be readily generalizable to arbitrarily large boards. It is plausible that the essence of Go would not be altered substantially if the game were played on a 29×29 board, instead of the traditional 19×19 one (for the definition of Go see below). And for these games the methods of complexity (in particular, **PSPACE**-completeness) come into play.

In the remainder of this section we show that two such generalized games are **PSPACE**-complete.

The Game of Geography

Geography is an elementary-school game played by two players, called here "I" and "II." Player I starts the game by naming a fixed starting city, say "ATHENS." Player II must then find a city whose name starts with the last letter of the city previously named, such as "SYRACUSA." Player I then must reply with something like "ALEXANDRIA" ("ATHENS," as well as any other city already played by one of the players, cannot be reused throughout the game). And so on. The first player that cannot play (presumably because all cities whose name starts with the last letter of the current city have been used) loses.

We can reformulate this game as follows: We have a directed graph $G = (V, E)$ whose nodes are all cities in the world, and such that there is an edge from city i to city j if and only if the last letter of the name of i coincides with the first letter of j. Player I picks the prespecified node 1, then player II picks a node to which there is an edge from 1, and so on, with the player alternating defining a path on G. The first player that cannot continue the path because all edges out of the current tip lead to nodes already used, loses.

We can generalize this *to any given graph G;* this generalization may imply not only a planet with arbitrarily many cities, but, even less realistically, one with an arbitrarily large alphabet. In fact, not all graphs represent a "starts with the same letter as the end of" relation. In any event, we are interested in the following computational problem:

GEOGRAPHY: Given a graph G and a starting node 1, is it a win for I?

Theorem 19.3: GEOGRAPHY is **PSPACE**-complete.

Proof: The game of GEOGRAPHY has the following two important properties:

(a) The length of any legal sequence of moves is bounded by a polynomial in the size of the input; in particular, the game must come to a conclusion after at most $|V|$ moves.

(b) Given a "board position" (that is, a graph, a path from node 1, and an indication whether I or II plays next) there is a polynomial-space algorithm that constructs all possible next moves and board positions; or, if there is none, decides whether the board position is a win for I or II.

Any such game can be solved in **PSPACE**. The algorithm is the same we used for QSAT: Given the input, we construct in polynomial space the "game tree," that is, the tree of board positions starting from the first. The leaves of this tree are evaluated to **true** or **false**, depending on whether they are a win or loss for I. Any non-leaf board position is considered as OR gates if I is about to move, and an AND gate if II is. We can avoid gates with more than two

inputs by replacing each such gate by a binary tree of enough gates of the same sort. All this can be done in polynomial space. We can then evaluate this tree also in polynomial space, to obtain the answer to the input.

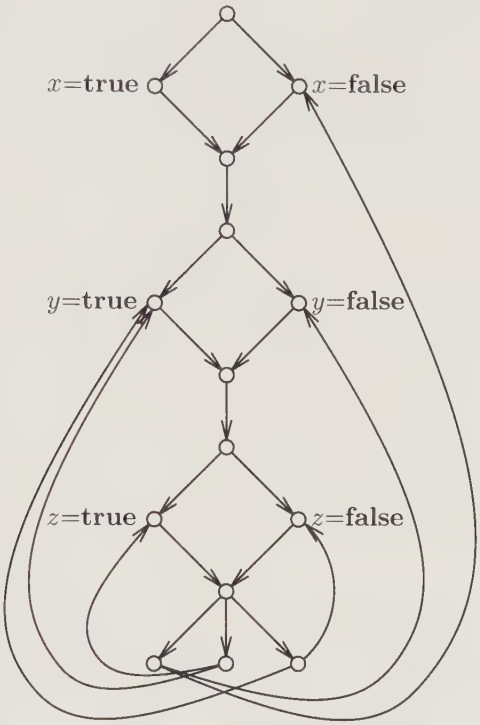


Figure 19-1. The reduction for GEOGRAPHY.

We shall now reduce QSAT to GEOGRAPHY. Suppose that we are given an instance of QSAT, say

$$\exists x \forall y \exists z [(\neg x \vee \neg y) \wedge (y \vee z) \wedge (y \vee \neg z)].$$

The construction for this example is illustrated in Figure 19.1; the generalization to any quantified expression is obvious. Each variable is replaced by a diamond-like "choice gadget," and all these gadgets are arranged in tandem. The starting node is the top of the diamond corresponding to the first variable. Any path from the starting node in this graph will pick one of the two sides of each diamond. We can think of such a path as a truth assignment for the variables, *where at the ith diamond the* **false** *literal is chosen.* That is, if the player

decides x to be **true** then the $\neg x$ side of the diamond is chosen, and vice-versa. Notice that this way, significantly, *I decides the existential variables and II the universal ones.* After this is done (assuming with no loss of generality that the last quantifier is universal) II picks a node corresponding to a clause, in an effort to argue that this clause is not satisfied by the chosen truth assignment (these are the bottom nodes). The only nodes available to I in the next move are the middle nodes of the diamonds corresponding to the literals in this clause. If there is no **true** literal in the clause, then I has no move and loses immediately. If there is a literal in the clause that satisfies it (that is, a literal that has not been picked by the path) then I moves to this literal and II loses at the next step.

Suppose that the resulting graph is a win for player I. This means that, no matter how II plays, player I can choose a path that leads II to a used city. But this means that I has a choice for the first diamond such that for all choices of II at the second diamond, and so on for the other diamonds, such that no matter which clause node player II chooses next, I has a choice of an unused literal. But this translates directly to a winning strategy for $\exists$ in the given instance of QSAT: $\exists$ has a choice for x_1 such that for both choices for x_2, etc., for all clauses there exists a satisfying literal. Hence the given instance of QSAT is **true**. The converse is also immediate. □

The Way to Go

Go is an ancient game whose board is a 19×19 grid of "points." Two points are considered adjacent if they are on the same row and adjacent columns, or vice-versa (but not if they abutt diagonally). Two players, Black and White, alternate placing a (respectively, black or white) "stone" on any unoccupied point; Black moves first. The objective of both players is, loosely speaking, *to form large, safe groups of their own color, and to capture as many stones of the opposite color as possible.* We explain what these terms mean next.

A *black group* is a connected component of the subgraph of the grid induced by the black stones; similarly for a white group (for example, in Figure 19.2 there are three black groups and five white groups). A black group is *surrounded* if no stone in it is adjacent to an unoccupied point. As soon as a black group becomes surrounded (presumably due to White's placing of a stone to its last unoccupied outlet) all black stones in it are captured by White, and removed from the board. Similarly for a white group surrounded by Black. For example, the black group in the left of Figure 19.2 is about to be surrounded and captured by White.

The ∞-shaped white group on the lower right of Figure 19.2 is "safe," that is, it is under no danger of being surrounded and captured by Black. The reason for this is that it includes two single holes (called "eyes") that cannot be

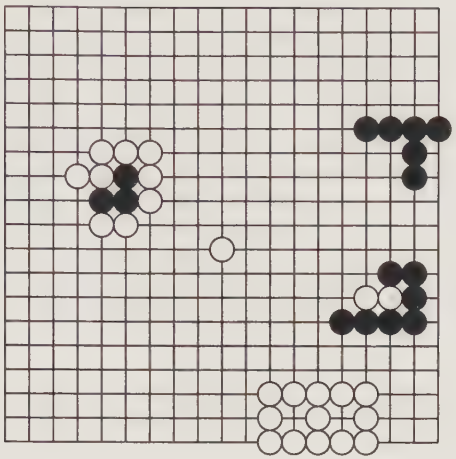


Figure 19-2. A Go position.

simultaneously filled by Black, and thus provide "permanent breathing room" for all white stones connected to them. Therefore, any white group such as the one on the upper right corner can be made safe if it is connected to the safe ∞-shaped group by a white path. Often a game of Go degenerates into a race between one player who tries to connect a large group to a safe group, through a path many parts of which may already be in place, and the opponent who tries to block this connection. Our proof that the $n \times n$ generalization of Go is **PSPACE**-complete will rely on such a race.

To simplify and enable our proof that Go is **PSPACE**-complete we must deviate a little from standard Go rules. First, we shall assume an arbitrarily large $n \times n$ board —this is a necessary ingredient of any complexity argument about board games. Second, we shall omit a number of complex rules that do not affect the kind of positions that are likely to come up in our proof. More importantly, notice that we have not determined when the game ends and who wins. We shall assume that the game ends after n^2 moves by the two players (that is, when they have both placed enough stones which, with no captures, would fill the board). (A player may pass at any time, but this counts as a move towards termination). The winner is the player that at the end has the most uncaptured stones on the board; in case of a tie, White wins, say[†].

We define GO to be the following problem:

[†] The actual termination rules of Go are quite a bit more complicated, even slightly ambiguous. For one, there is no known *a priori* upper bound for the number of moves, such as the n^2 bound imposed by our rules, and thus we do not know if a more faithful generalization of Go to $n \times n$ boards would be in **PSPACE**.

GO: We are given an $n \times n$ board configuration after $k < n^2$ moves, with several black and white stones. It is Black's turn to play. Is it a win for Black?

A disclaimer about the significance of complexity results about problems such as GO above is perhaps in order. The fact (soon to be proved) that problem GO is **PSPACE**-complete simply means that, unless **P** = **PSPACE**, there is no polynomial-time algorithm that decides whether an arbitrary Go position is a win for White. For all we know, the most important Go position of them all, *the empty board*, may be an easy win for Black, and the kinds of positions that we prove hard never come up in optimal play. Having said this, let us prove

Theorem 19.4: GO is **PSPACE**-complete.

Proof: That the generalized game with simplified rules is in **PSPACE** follows from the general argument presented in the beginning of the proof of Theorem 19.3.

To show completeness, we shall reduce GEOGRAPHY to GO[‡]. In reducing GEOGRAPHY to GO we have to take advantage of the special structure of the GEOGRAPHY graphs produced in the proof of Theorem 19.3; in some sense, this is the continuation of a reduction from QSAT to GO, with GEOGRAPHY being just an intermediate byproduct.



Figure 19-3. Degree reduction in GEOGRAPHY.

Recall the GEOGRAPHY graph produced in the proof of Theorem 19.3 (Figure 19.1). It has a very specialized structure. If one ignores the "back" edges connecting the bottom nodes back to the literals, this graph is *bipartite*, that is, its set V of nodes are partitioned into two sets $V_{\rm I}$ and $V_{\rm II}$, such that I always plays when the current city is in $V_{\rm I}$, and II from $V_{\rm II}$. $V_{\rm I}$ consists of the top and

‡ No, we do *not* mean by deleting the letters E, G, R, A, P, H, and Y...

bottom nodes of the even-numbered diamonds, and the middle (literal) nodes of the odd-numbered diamonds; V_{II} contains all other nodes. We can easily modify the graph so that each node either has indegree one and outdegree at most two, or vice-versa (this is done by the replacements shown in Figure 19.3). Finally, we can make the graph *planar*. Consider any two crossing edges. A look in Figure 19.1 says that they are both *back edges*, and therefore *at most one of them is going to be traversed* in any play of the game. With this in mind we replace this crossing as in Figure 19.4. It is easy to see that once node a is played by I, the optimal play by both players continues along the path $(1, 3, 4, 6, b)$, and thus this replacement correctly simulates the two replaced edges. Playing 5 by I, or 8 by II, results in immediate defeat.

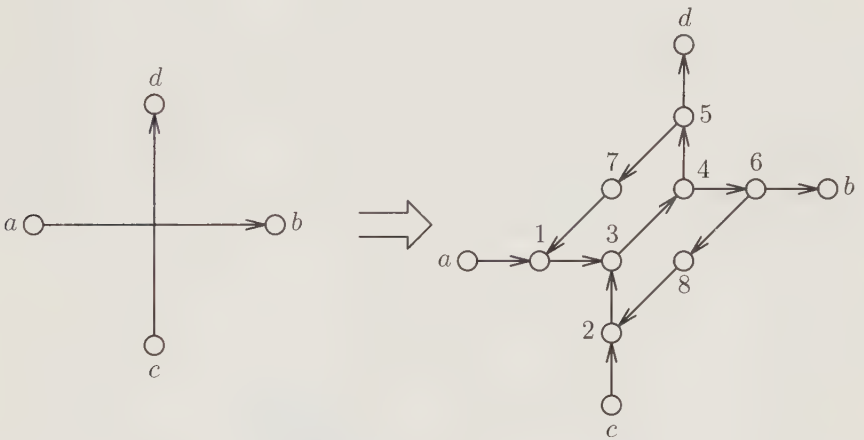


Figure 19-4. Crossing edges in GEOGRAPHY.

The nodes of the resulting graph fall into one of the following five categories:

(a) Decision nodes for player I (Figure 19.5(a)). These are the top nodes of the even-numbered diamonds, the clause nodes (expanded as in Figure 19.3), as well as the forced decision nodes introduced in Figure 19.4.

(b) Decision nodes for II (Figure 19.5(b)). These are the top nodes of the odd-numbered diamonds, and the bottom node of the bottom diamond (rather, the nodes to which it was expanded as in Figure 19.3).

(c) Merge nodes (Figure 19.5(c)). These are the bottom nodes of all diamonds, as well as one node in Figure 19.4.

(d) Test nodes (Figure 19.5(d)). These are the middle nodes of all diamonds, expanded as in Figure 19.3. The last act of the game is played at one of these.

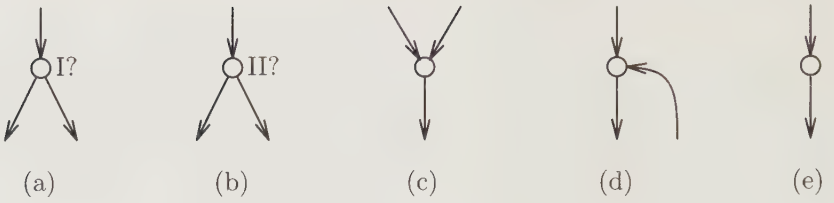


Figure 19-5. Five kinds of nodes.

(e) Finally we have trivial nodes (Figure 19.5(e)), with both indegree and outdegree one (the bottom nodes belong here, as well as certain nodes introduced in Figures 19.3 and 19.4).

The Go position that we are going to build has the following structure: The board is an $n \times n$ grid, where n is large enough ($n = 20|V|$ would do). We are at the $(n^2 - n)$th move—that is, we have n moves to go. A large part of the grid is occupied by a large white group that is almost surrounded by a smaller black group (see Figure 19.6). The white group is so large (and the number of remaining moves so small) that the whole game depends on whether the white group is captured, or whether it successfully connects to one of the many small safe white groups that are sprinkled on the board in a way that we describe next.

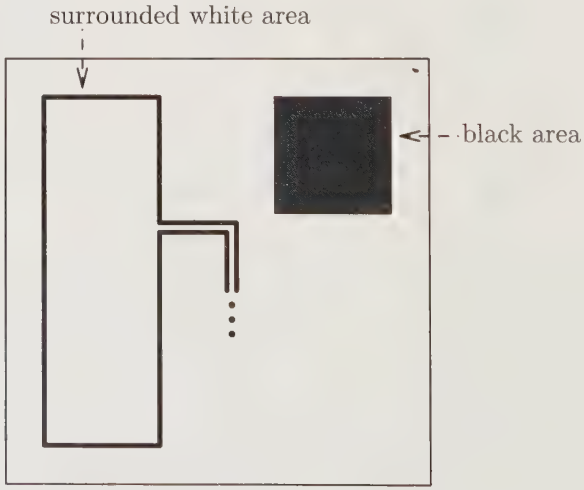


Figure 19-6. General structure of the position.

It is White's turn to play. The white group is completely surrounded,

except for a narrow "pipe" leaving the bulk of the group (see Figure 19.6). The pipe is also surrounded by the black group, but it does lead to a small "breath of air" (one or two unoccupied points). This pipe will lead to structures that simulate the five kinds of nodes of the GEOGRAPHY graph (the beginning of the pipe simulates the start node). In fact, the pipes that simulate the edges, and the various structures that simulate the nodes of the graph form an "embedding" of the GEOGRAPHY graph on the $n \times n$ grid. To implement this embedding, it will be necessary to allow the pipe to "bend" as shown in the Figure (it is easy to see that any planar graph can be embedded on a grid so that its edges are piecewise horizontal and vertical line segments). The idea is that there is a strategy for White which guarantees that the pipe will lead to a safe white group if and only if the original instance of GEOGRAPHY is a win for player I.

(a) Player I decision nodes are simulated by the Go position in Figure 19.7(a). When the players move in this part, the pipe from the top is already connected to the large white group, and it is White's turn to play. If I does not place a white stone at one of the positions 1 and 2 in the Figure, then Black wins: Black plays at 1 (or at 2, if White's move was at 5), this forces White to play at 2 (respectively, 1), at which point Black completes his surrounding of the White group by playing at 5 (respectively, 3). Hence, I must play at either position 1 or 2, and this simulates I's choice of the right or left edge out of the current node in GEOGRAPHY, respectively. (Notice the reversal of roles: If I takes the left edge in GEOGRAPHY, White will take the *right* pipe in GO.) If I plays at 1 then II responds at 2 closing off the other pipe. White must respond at 3, Black must continue at 4 (otherwise I will connect to the ∞-shaped group and win), and the play in this structure has been completed. Symmetrically if I plays 2 Black responds by 1, then White plays 5 and Black 6. Notice that it is again White's turn to play as the game is taken to the next structure (and so our hypothesis that White starts is preserved).

(b) Player II decision nodes are simulated by the structure in Figure 19.7(b). The only difference is that White must now move first to point 0, and Black has a choice between 1 and 2.

(c) Merge nodes are simulated as shown in Figure 19.7(c). The white group has been connected to either the left or the right pipe. Depending on which pipe the white group is connected, White must now play at 1 or 2 to further connect it to the downwards pipe, and Black must respond by playing at the point left unoccupied by White (otherwise Black loses immediately).

(d) Test nodes are simulated as shown in Figure 19.7(d). Recall that these nodes are potentially visited twice by the players: Once right after the corresponding decision node, and once at the end. The first time, as soon as the upper pipe is connected to the white group White must play 1,

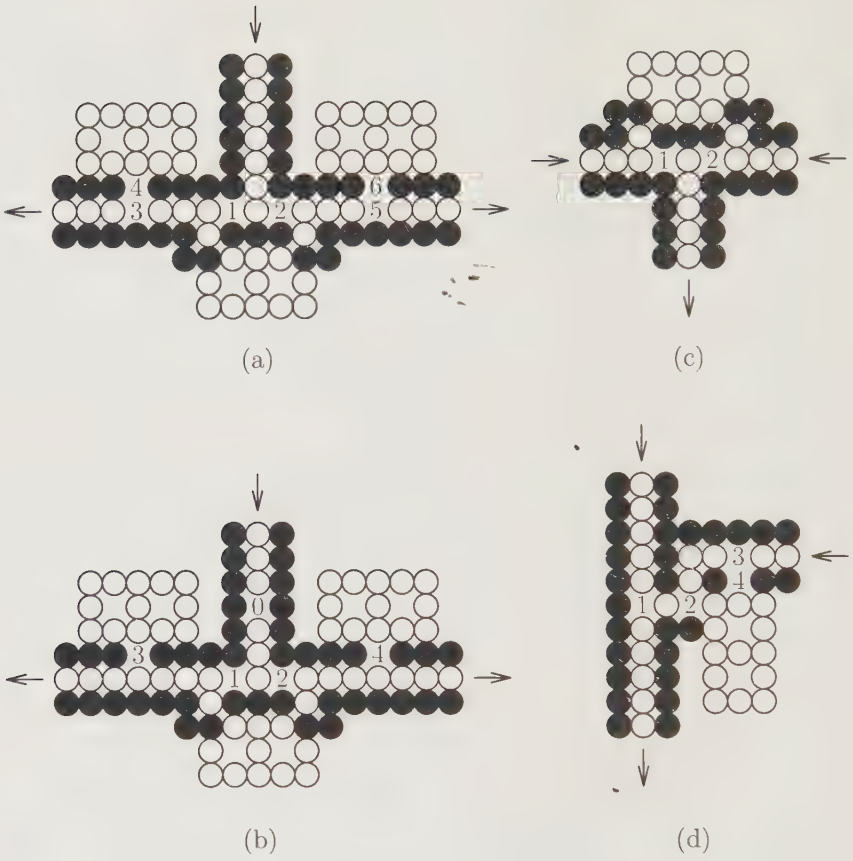


Figure 19-7. Node structures.

Black must respond with 2 (otherwise White will connect to the vertical ∞-shaped group), and the play proceeds to the next structure down the pipe. If at the end the play ends up in this structure through the pipe in the right, White must play 3. At this point, if the play has already passed through this structure (that is, if it corresponds to an unused city in GEOGRAPHY, or a **false** literal in QSAT) then there is a black stone at 2, and Black wins at 4. If not, Black cannot cover both 2 and 4, and White connects to the vertical ∞ group and wins at the next move.

(c) Trivial nodes with indegree and outdegree one are not represented in our construction; a pipe simulates an arbitrarily long chain of such nodes. The only purpose of these nodes in GEOGRAPHY was to control whose turn it is to play; in our instance of Go, White always plays first.

It follows from our discussion of the various components of the Go position that the constructed position is a win for White (that is, White has a strategy for winning in the face of any possible strategy of Black) if and only if the given instance of GEOGRAPHY is a win for player I (which is true if and only if the QSAT instance is a win for player $\exists$). □

19.2 GAMES AGAINST NATURE AND INTERACTIVE PROTOCOLS

Consider the following stochastic scheduling problem: We are given a directed acyclic graph $G = (V, E)$, where the nodes are *tasks* to be executed on two processors (see Figure 19.8). The two processors are identical, and any task can be executed on either one. A task can start on a processor if *any* of its predecessors has finished. That is, the predecessors in this scheduling problem are alternative prerequisites, of which at least one has to be executed; this is the first idiosyncracy that sets this problem apart from more conventional scheduling problems. Second, there is a subset M of V that contains the *mandatory tasks*. Only these tasks have to be executed; the remaining tasks can be executed if needed as prerequisites (tasks in M are shown as solid dots in the Figure). Third, and perhaps more important, the execution time of any task (mandatory or not) on any processor is *a unit Poisson random variable*. That is, for each $t \geq 0$, the probability that the execution time is at most t is equal to $1 - e^{-t}$; the execution times of different tasks are independent. We seek a scheduling strategy that will minimize the expected elapsed time until the completion of the last mandatory task.

Let us determine what constitutes a scheduling strategy in this context. At any point there are several tasks that can be scheduled (that is, either they have no predecessors in G, or at least one predecessor has been completed). Any reasonable strategy would pick two of these tasks (assuming there are two or more of them) and schedule them; if there is only one such task, there is no choice but to schedule it by itself on the machine. Suppose now that one of the two tasks that are being executed actually completes. The expected elapsed time before this event happens is exactly $\frac{1}{2}$ (the minimum of two unit Poisson random variables). Because of the *memoryless property* of the Poisson distribution, it is easy to see that the probability that the other task will require *additional* execution time at most t is *still* $1 - e^{-t}$. In other words, the execution time distribution of the task has not been affected by the fact that it has executed for as long as it took for the other task to complete. It is therefore not suboptimal to *suspend the execution of the second task* and start a new round. We must now decide which two of the tasks that are now available, *including the suspended one*, to schedule next. The time period between two consecutive

such decisions is called a *decision cycle*. If during a decision cycle only one task is being executed, then its expected time time to completion is 1.

Remember that we wish to minimize the expected elapsed time until the last mandatory task has been completed. This expectation can be easily calculated: It is precisely

$$\frac{1}{2}T_2 + T_1,$$

where T_2 is the total number of decision cycles during which two tasks are being executed, and T_1 the number of decision cycles during which only one task is available for execution. We wish to design a strategy (that is, a function which, given any possible subgraph of the given task graph having at least two available tasks, selects two available tasks for execution) that minimizes the expectation of this figure. We can define the following computational problem:

STOCHASTIC SCHEDULING: Given a task graph $G = (V, E)$, a set of mandatory tasks $M \subseteq V$ and a rational number B, is there a scheduling strategy such that the expectation of $\frac{1}{2}T_2 + T_1$ is less than B?

This is a typical problem of *decision-making under uncertainty*. We are repeatedly faced with a decision (e.g., which tasks to schedule), followed by a random event (in our example, which of the two tasks finishes first), followed by a new decision, a new random event, and so on. The outcome of the process depends on both decisions and random events; we wish to design a strategy that optimizes the outcome.

Problems of decision-making under uncertainty can be considered as a special kind of game against a randomized opponent: A "game against Nature." The framework, in terms of a "board," number of moves, etc., is exactly as with ordinary games. The difference is now that one player strives to win, while the other player is uninterested in winning, *and plays at random*. This at first sight may seem a most favorable situation, and by implication a computationally easy one. This is not so. Our goal in games against Nature is to find a *strategy that maximizes our probability of winning* (or some other expected reward). This turns out to be computationally as hard as playing against an optimizing opponent!

Naturally enough, there is a variant of satisfiability that captures this situation:

SSAT (for *stochastic satisfiability*): Given a Boolean expression ϕ, is there a truth value for x_1, such that if the value of x_2 is selected at random, there is a truth value for x_3, etc., such that the probability that ϕ is finally satisfied is greater than $\frac{1}{2}$? One can write this as

$$\exists x_1 \mathsf{R} x_2 \exists x_3 \mathsf{R} x_4 \ldots \mathbf{prob}[\phi(x_1, \ldots, x_n) = \mathbf{true}] > \frac{1}{2},$$

where we have used a new kind of quantifier Rx, pronounced "for random x," with obvious meaning.

One can define a natural variant of alternating polynomial time that captures problems of decision-making under uncertainty, as follows:

Definition 19.1: A *probabilistic alternating Turing machine* is a precise alternating polynomial time Turing machine M, all the computations of which on input x have equal length $2|x|^k$, and the number of nondeterministic choices is uniformly two. Furthermore, the computation strictly alternates between states in two disjoint sets, which we shall now call K_+ and K_{MAX} (instead of the usual K_{OR} and K_{AND}).

Consider a configuration C in a computation of the probabilistic alternating Turing machine M on input x. The *acceptance count* of configuration C is defined as follows: If the state of C is an accepting state, then its count is one; if it is a rejecting state its count is zero. Otherwise, if the state of C is in K_+, then the acceptance count of C is the *sum* of the acceptance count of the two successor configurations. Finally, if the state of C is in K_{MAX}, then the acceptance count of C is the *maximum* between the two counts. We say that the probabilistic alternating Turing machine M accepts x if the acceptance count of the initial configuration is more than $2^{|x|^k-1}$.

To put it otherwise, M accepts x if for each configuration C with state in K_{MAX} there is a choice of one of the two successor configurations, such that, if we consider the resulting computation tree with $2^{|x|^k}$ leaves (recall that the original tree had height $2|x|^k$) a majority of these leaves is accepting.

We denote the class of all languages decided by probabilistic alternating polynomial-time Turing machines **APP** (it is an alternation-based version of **PP**). □

The following may come as a slight surprise:

Theorem 19.5: APP = PSPACE.

Proof: In one direction, we show that acceptance by a probabilistic alternating machine can be decided in polynomial space. Our algorithm keeps a counter of the accepting leaves seen so far, initialized to zero; we also maintain the configuration currently visited. If an accepting configuration is visited then the counter is increased by one, while at rejecting configurations the counter is not increased; we then return to the predecessor configuration (easy to calculate from the current one). If a K_{MAX} configuration is reached for the first time, our algorithm nondeterministically selects one of the two successor computations (remember that, by Savitch's Theorem, nondeterminism is free in the context of polynomial space) and continues from there. When a K_{MAX} configuration is reached from one of its successors (the algorithm has evaluated the whole subtree), we continue our ascend to its own predecessor. If a K_+ configuration is reached for the first time, then the algorithm visits next its first successor;

when it returns after visiting the first subtree, the algorithm visits the second successor; and the third time it returns to the predecessor. The input is accepted if the counter is larger than $2^{|x|^k-1}$. It is clear that only the counter (polynomially many bits) and the current configuration need be maintained.

The other direction is an elaboration of the proof that **NP** $\subseteq$ **PP** (Theorem 11.3). Suppose that L is decided by an alternating Turing machine M in time n^k. We can design a probabilistic alternating machine M' that decides L, as follows: M' at its first step (from a K_+ state) branches to two states: One that resumes the ordinary computation of M on the input, and another that has a *single accepting computation* and the same polynomial number of steps. Suppose that there is a way of assigning a successor to each K_{MAX} configuration of the computation tree of M' so that the majority of leaves in the resulting tree is accepting. Since only one accepting leaf can come from the second half of the leaves (those corresponding to the second branch from the start state), this means that all leaves in the first half (those reflecting the computation of M) must be accepting, and thus the input is in L. The converse is also easy. □

The following result is rather immediate now:

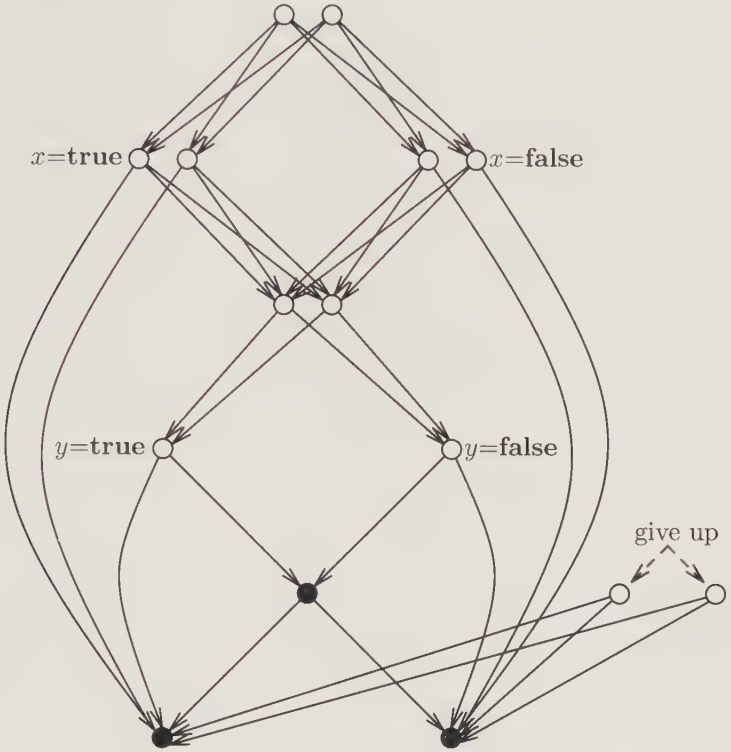
Theorem 19.6: SSAT is **PSPACE**-complete.

Proof: A probabilistic alternating polynomial-time Turing machine that decides SSAT guesses the truth values for the variables, only that the existentially quantified variables are guessed at a state in K_{MAX}, while the randomly quantified variables are guessed at a state in K_+. Strict alternation can be achieved by interjecting dummy states.

To show completeness, consider any language L in **PSPACE**, decided by the probabilistic alternating Turing machine M, and an input x. By Cook's Theorem we can construct an expression that captures the computation of M on x. All we need to do in order to reflect the acceptance convention of probabilistic alternating Turing machine is to quantify the variables that stand for nondeterministic choices from a state in K_+ by a random quantifier, and all other variables by an existential quantifier. Naturally, quantifiers corresponding to nondeterministic choices are ordered in accordance to the temporal order of these choices. □

Theorem 19.7: STOCHASTIC SCHEDULING is **PSPACE**-complete.

Proof: STOCHASTIC SCHEDULING can be decided by the following probabilistic alternating machine: The choice of the two tasks to be scheduled next can be simulated by a sequence of $2 \log |V|$ nondeterministic choices from K_{MAX} states, each guessing a bit of the two chosen tasks (with intermittent dummy K_+ configurations to abide by the alternation requirement of these machines). The random choice of one of the two tasks to finish first is simulated by a K_+ configuration.



$$\exists x \mathbf{R} y[(x \vee y) \wedge (\neg x \vee \neg y)]$$

Figure 19-8. Stochastic scheduling.

There is a difficulty in that in the STOCHASTIC SCHEDULING problem we accept if the expected time is less than the given bound B, while probabilistic alternating machines accept if the probability of acceptance is more than half. This can be corrected as follows: Each computation of the machine described so far will end up with an actual schedule, with cost $T = \frac{1}{2}T_2 + T_1$. From each such configuration we make a sequence of $\log |V| + 1$ nondeterministic choices from K_+ states (interleaved as usual with dummy K_{MAX} states) so that the total number of accepting leaves in the resulting subtree is exactly $|V| + 2B - 2T$ out of the total of $2|V|$—here we are assuming without loss of generality that $|V|$ is a power of two. If we fix a choice for the K_{MAX} configurations (that is, a scheduling strategy), the probability of acceptance is now precisely $\frac{|V|+2B-2\overline{T}}{2|V|}$, where $\overline{T}$ is the expected completion time for the scheduling strategy chosen. It

is immediate that this probability is greater than $\frac{1}{2}$ if and only if $\overline{T} < B$, as required.

To show that STOCHASTIC SCHEDULING is **PSPACE**-complete we shall reduce SSAT to it. We sketch the reduction below (see Figure 19.8 for an example). Given an instance of SSAT with n variables and m clauses, the task graph G (superficially very similar to the graph constructed for GEOGRAPHY) consists of a stack of structures corresponding to the variables, plus a task for each clause. The structure corresponding to existential variables is a diamond (the top one), in which the best strategy is to select one side and schedule both tasks in it; this has the effect of selecting a truth assignment for the corresponding variable. The structure corresponding to a randomized variable (the second from the top) is even simpler: Now the optimum strategy is to schedule both sides (leaving to Nature to decide on a truth assignment). The only mandatory tasks are the bottom node of the stack (so it is necessary to schedule a path through the stack), plus the clause tasks. Now each clause task is preceded by all literals appearing in it, plus by two other non-mandatory tasks called the "give-up tasks." Remember that for a task to be scheduled *any one* (not all) of its predecessors must have been completed. Once a path through the stack has been completed (in $2n+1$ cycles during all of which the machines have been fully busy) there are two options: If the resulting truth assignment satisfies all clauses, then we can schedule the bottom task and all m clause tasks in $m+1$ more cycles, of which only the last will be idle (total expected elapsed time: $\frac{1}{2}T_2 + T_1 = \frac{1}{2}(2n+1+m)+1 = \frac{2n+m+3}{2}$). Otherwise, we can schedule the give-up tasks first, and, after one of them finishes, schedule the clauses, at the expense of an extra busy cycle (total elapsed time $\frac{2n+m+4}{2}$). It follows that, if p is the probability that a given strategy for selecting truth assignments for the existential variables satisfies all clauses, then the expectation of the corresponding scheduling strategy is $p\frac{2n+m+3}{2} + (1-p)\frac{2n+m+4}{2}$, which is less than $B = \frac{4n+2m+7}{4}$ if and only if $p > \frac{1}{2}$. □

There are several more problems of decision-making under uncertainty that are **PSPACE**-complete (see the references).

Interactive Protocols

In Chapter 11 after defining **PP** we immediately backed off to define a weaker class **BPP** with similar definition, but with the additional requirement that the probability of acceptance or rejection must be bounded away from $\frac{1}{2}$ (at least $\frac{3}{4}$ for acceptance, at most $\frac{1}{4}$ for rejection). We can similarly relax the definition of class **APP** (which we just proved equivalent to **PSPACE**) to define a similar *alternating* class, call it **ABPP**. For a language in **ABPP** there is a probabilistic alternating Turing machine such that if input x is in L then there is a set of choices of successor configurations for the K_{MAX} configurations in

the computation tree such that at least $\frac{3}{4}$ of the leaves in the resulting tree are accepting. In contrast, if $x \notin L$, then for all choices of successors for the K_{MAX} configurations at most $\frac{1}{4}$ of the leaves of this tree are accepting.

There is a close relation between **ABPP** and **IP**, the class of problems decidable by *interactive protocols* (recall Section 12.2). Any **ABPP** computation can be thought of as an interactive protocol, and in fact one in which *Alice can see the random bits of Bob* (the random bits used so far can be deduced from the current node on the tree of nondeterministic computations). Of course the error probability is $\frac{1}{4}$, instead of $2^{-|x|}$ we require of interactive protocols, but never makes a difference (recall the discussion in Section 11.3). So, **ABPP** $\subseteq$ **IP**. In turn, **IP** is a subset of **PSPACE**: In polynomial space we can examine, one after the other, all possible interactions between Bob and Alice, and calculate the probability of acceptance. Hence we arrive at the following chain of inclusions:

Proposition 19.1: **ABPP** $\subseteq$ **IP** $\subseteq$ **APP** $=$ **PSPACE**. □

In the realm of nondeterministic computation, probabilistic computation by clear majority, captured by **BPP**, seemed substantially weaker than the sharp-cutoff version, **PP**. It would be remarkable if **BPP** includes **NP** (as **PP** does), and thus all **NP**-complete problems have credibly practical randomized algorithms; such an eventuality is considered highly unlikely. *Very surprisingly*, in the world of alternation sharp cutoffs and robust cutoffs are equivalent, and the chain of inclusions in Proposition 19.1 collapses! The hard part is proving the upper equality (the lower one follows as a corollary of the proof):

Theorem 19.8 (Shamir's Theorem): **IP** $=$ **PSPACE**

Proof: One of the inclusions follows from Proposition 19.1. For the other inclusion, since **IP** is obviously closed under reductions, it suffices to show that the **PSPACE**-complete problem QSAT is in **IP**. We shall do this by describing a clever interactive protocol that decides QSAT. Suppose that Alice and Bob are given a quantified Boolean expression ϕ, say

$$\phi = \forall x \exists y (x \vee y) \wedge \forall z ((x \wedge z) \vee (y \wedge \neg z)) \vee \exists w (z \vee (y \wedge \neg w)). \qquad (1)$$

Notice that, for reasons that will become clear immediately, we do not assume that the expression is in prenex form. However, expression (1) has another useful property: *No occurrence of a variable is separated by more than one universal quantifier from its point of quantification.* We call such expressions *simple*. It turns out that it is not a loss of generality to assume that the given expression is simple:

Lemma 19.1: Any quantified Boolean expression ϕ can be transformed in logarithmic space to an equivalent simple expression.

Proof: Consider a universal quantifier $\forall y$ and a variable x quantified before $\forall y$ and used after $\forall y$. That is, ϕ is of the form $\ldots Qx \ldots \forall y \psi(x)$. We transform

ϕ as follows:

$$\ldots Qx \ldots \forall y \exists x' ((x \wedge x') \vee (\neg x \wedge \neg x')) \wedge \psi(x').$$

That is, we introduce a new existentially quantified variable x' which is going to be the new name of x, and postulate in disjunctive normal form that x and x' must be equal. Hence, for each universally quantified variable y, starting at the front of the expression, we modify ϕ as shown above for each other variable x that is quantified before y and used after y. Since for each y we have to do this a number of times bounded by the original number of variables in ϕ, it follows that after $\mathcal{O}(n^2)$ steps the resulting expression is simple. □

Applied to a simple expression ϕ, the protocol works by first converting ϕ into a roughly equivalent *arithmetic* expression, and then having Alice convince Bob that this *arithmetization* of ϕ is non-zero. We can assume that in ϕ negation is applied only to variables, not whole subexpressions. If not, then we can "propagate" any negation sign through the other logical connectives and quantifiers, until it reaches the level of variables, recall the reduction to MONOTONE CIRCUIT VALUE in the corollary to Theorem 8.1; this has to be done before the conversion of the expression to a simple one, described in the previous paragraph. To arithmetize ϕ we replace its Boolean variables with integer variables, with the convention that $x = 0$ means that x is **false**, and $x = 1$ means that x is **true**. $\vee$ is thus replaced by $+$, and $\wedge$ by $\times$. As a result, for whole expressions **false** translates to 0, while **true** translates to any positive value. $\neg x$ is simulated by $1 - x$ (recall that, very conveniently, we do not have negation of general expressions). Quantifying existentially an expression with respect to a variable x now corresponds to *adding* the values of the arithmetization of the expression, for $x = 0, 1$. Similarly, universal quantification corresponds to taking the product of these two values. For example, the arithmetization of ϕ in (1) above is this:

$$A_\phi = \prod_{x=0}^{1} \sum_{y=0}^{1} [(x + y) \cdot \prod_{z=0}^{1} [(x \cdot z + y \cdot (1 - z)) + \sum_{w=0}^{1} (z + y \cdot (1 - w))]]. \quad (2)$$

The scope of the summation and product signs extends to the end of the expression. We call such expressions Σ-Π *expressions*. Notice that, since ϕ is a fully quantified expression with no free Boolean variables, A_ϕ has no free variables either, and so its value is a nonnegative integer. For example, A_ϕ above evaluates to 96. In fact, we should have expected that the value is non-zero, because ϕ is **true**. This important property of arithmetization is captured by the following lemma:

Lemma 19.2: For any quantified expression ϕ with connectives $\wedge$, $\vee$ and negation only over variables, ϕ is **true** if and only if $A_\phi > 0$.

Proof: We shall prove by induction on the structure of ϕ a slightly stronger statement:

For any expression ϕ and any truth assignment to its free variables, if the truth value of ϕ is **true** *then A_ϕ, with its free variables taking values according to the obvious correspondence between* **true-false** *and* 1-0, *evaluates to a positive integer; if ϕ is* **false**, *then the value is zero.*

The statement certainly holds for literals. If $\phi = \psi_1 \vee \psi_2$, then ϕ is **true** if and only if at least one of ψ_1 and ψ_2 evaluates to **true**, which by induction happens if and only if at least one of A_{ψ_1}, A_{ψ_2} evaluates to a positive number, which finally happens if and only if $A_\phi = A_{\psi_1} + A_{\psi_2}$ evaluates to a positive integer (we know by induction that A_{ψ_1} and A_{ψ_2} are nonnegative integers). The induction steps for $\wedge$, $\forall x$ and $\exists x$ are very similar and are omitted. □

Thus, all Alice has to convince Bob is that a given Σ-Π expression evaluates to a positive integer. As a first step, Alice may wish to send this integer to Bob (she can certainly compute it with her exponential computing powers). But there is a difficulty immediately: *This number may be too large.* For example, the Σ-Π expression

$$\prod_{x_1=0}^{1} \prod_{x_2=0}^{1} \cdots \prod_{x_k=0}^{1} \sum_{y_1=0}^{1} \sum_{y_2=0}^{1} (y_1 + y_2) \tag{3}$$

evaluates to 4^{2^k}—a number with exponentially many bits and thus impossible to compute and transmit. The solution to this problem is is provided by the following lemma.

Lemma 19.3: If the value of Σ-Π expression A, of length n, is non-zero, then there is a prime p between 2^n and 2^{3n} such that $A \neq 0 \bmod p$.

Proof: Suppose that $A = 0 \bmod p$ for all primes in this range. By the Chinese remainder theorem (Corollary 2 to Lemma 10.1), it is 0 modulo their product. *We shall argue that this product is larger than the value of A*, and hence the true value is 0, contradicting our hypothesis that ϕ is **true**.

First, it is easy to see that the value of A is at most 2^{2^n}: Each additional operation (addition, multiplication, summation, or product) at most *squares* the value. (In fact, only multiplications and products may actually square the value; the other operations at most double it.) Since there are at most n operations in A, we conclude that the value is no larger than 2^{2^n}.

We shall next show that the number of different primes between 2^n and 2^{3n} is at least 2^n, and hence their product is larger than 2^{2^n}; this would complete the proof of the lemma. This follows from the prime number theorem (an important and deep result about the distribution of primes, see the discussion in Problem 11.5.27); but it is also implied by the following simple calculation:

Claim: The number of primes up to n is at least $\sqrt{n}$.

Proof of the claim: If a number $i \leq n$ is not divisible by any prime smaller than $\sqrt{n}$, then i is a prime. Now, of the numbers that are less than or equal to n, at most one-half are divisible by two. Of the remaining ones, at most one-third are divisible by three. And so on for all primes up to $\sqrt{n}$. It follows that the number of primes up to n is at least $n \prod_{p \leq \sqrt{n}} \frac{p-1}{p}$, with the product ranging over all primes up to $\sqrt{n}$, which is at least as large as $n \prod_{i=2}^{\lfloor \sqrt{n} \rfloor} \frac{i-1}{i} \geq \sqrt{n}$. □

According to the lemma, the whole protocol whereby Alice convinces Bob that the Σ-Π expression A evaluates to something other than zero can be conducted modulo a prime p, of Alice's choice, where $2^n \leq p \leq 2^{3n}$. Let us now describe this protocol in detail, using the Σ-Π expression A in equation (2) as our running example.

Alice starts by sending Bob the prime number p modulo which all computations are to be conducted, together with its primality certificate —assume this number is 13 (ignoring for a moment the requirement that it be at least 2^n). She also sends the value a of A modulo p, which she computed using her exponential powers —in our example $a = 96 \bmod 13 = 5$.

The computation proceeds in stages, one stage for each Σ and Π symbol in A. At the beginning of each stage there is a current Σ-Π expression A starting with a Σ_x or a Π_x, and its alleged value a modulo p, supplied by Alice. If the first Π or Σ is deleted, then the resulting expression is a polynomial in x, call it $A'(x)$. *Bob demands from Alice the coefficients of this polynomial* (which are hard to compute, but not for Alice). Now, in general the degree of this polynomial can be exponential in n, because of repeated squaring (consider the Σ-Π expression (3) above with x_1 replacing y_1 in the last parenthesis). However, since we have assumed that our original expression ϕ was simple, it follows that $A'(x)$ can be rewritten so that it has only one Π symbol—the others are products over quantities that do not involve x and are therefore constants. It follows that the degree of $A'(x)$ is at most $2n$, since any other symbol except for Π can increase the degree of $A'(x)$ by at most one. Therefore, Alice has no problem transmitting the coefficients of $A'(x)$ to Bob.

In our example the polynomial $A'(x)$ is $2x^2 + 8x + 6$. Once Bob receives it, he verifies that $A'(0) \cdot A'(1) = a \bmod p$ (in our example, $6 \cdot 16 = 5 \bmod 13$), and hence the polynomial presented is consistent with the alleged value a. Bob now wants to delete the leading Π_x symbol and continue the verification with a new, smaller Σ-Π expression A. But, if he does so, the remaining expression has x as a free variable and so is not an evaluation problem of the same nature. To convert it into an evaluation problem with no free variables, *Bob replaces x with a random value modulo p*, say 9. The resulting Σ-Π expression is

$$\sum_{y=0}^{1}[(9+y)\cdot\prod_{z=0}^{1}[(9\cdot z+y\cdot(1-z))+\sum_{w=0}^{1}(z+y\cdot(1-w))]].$$

Since this is just $A'(9)$, and $A'(x)$ was claimed by Alice to be $2x^2+8x+6$, if Alice is correct then the value of the new A must be $a = 2\cdot 9^2+8\cdot 9+6 = 6 \bmod 13$. We can therefore start a new round.

$A'(y)$ is again A with the leading Σ deleted. Alice evaluates it to be $A'(y) = 2y^3+y^2+3y \bmod 13$, and sends this information to Bob. Bob checks that $A'(0)+A'(1) = 6 \bmod 13$. He can now delete the leading Σ and replace y with a random number modulo 13, say $y = 3$. The claimed value of the new expression is therefore $A'(3) = 7 \bmod 13$. Now the new expression starts with the factor $(9+3) = 12 \bmod 13$. If 12 times the rest of the expression gives us 7 mod 13, it must be that the rest of the expression is $7\cdot 12^{-1} \bmod 13$, and even Bob can run Euclid's algorithm to determine that $12^{-1} = 12 \bmod 13$. It follows that the claimed value of the rest of the expression

$$A = \prod_{z=0}^{1}[(9\cdot z+3\cdot(1-z))+\sum_{w=0}^{1}(z+3\cdot(1-w))]$$

is $a = 7\cdot 12 = 6 \bmod 13$. We are ready to start the next round.

Deleting the leading Π from A we have $A'(z)$, which Alice claims to be $A'(z) = 8z+6$. Bob checks that $A'(0)\cdot A'(1) = 6 = a \bmod 13$. He then generates a random value for z, say $z = 7$. The new expression, after deleting the leading Π from A, must have value $A'(7) = 10 \bmod 13$. This expression starts with the term $(9\cdot 7+3\cdot(1-7)) = 6 \bmod 13$. Since the whole expression is supposed to have value 10, the value of the remaining expression

$$A = \sum_{w=0}^{1}(7+3\cdot(1-w))$$

must be $a = 10-6 = 4$. Alice now claims that $A'(w) = 10-3\cdot w$. Bob checks that $A'(0)+A'(1) = a = 4 \bmod 13$. *Finally, since there are no more* Σ *and* Π *symbols in* $A'(w)$*, Bob can check by himself whether the claimed* $A'(w)$ *is correct.* It is, and Bob is convinced that the Σ-Π expression (2) is indeed non-zero modulo p, and hence non-zero; equivalently, that the quantified Boolean expression ϕ in (1) is **true**.

It should be clear that, if the true value of A is non-zero, then the protocol convinces Bob of this. It remains to show that, if $A = 0$, no efforts by Alice can convince Bob with more than negligible probability. The proof is simple: We claim that, if $A = 0$ and still Alice starts with a non-zero value a, then *with probability* $(1-\frac{2n}{2^n})^{i-1}$ *the value of* a *claimed at the* i*th round is wrong.*

The first value a claimed by Alice is non-zero and thus certainly wrong; hence the claim is true when $i = 1$. By induction then, for $i > 1$ we know that the $i-1$st value was wrong with probability $(1 - \frac{2n}{2^n})^{i-2}$. Suppose that it was indeed wrong. Since the polynomial $A'(x)$ produced by Alice in the ith round must be such that $A'(0) + A'(1)$ (or $A'(0) \cdot A'(1)$, depending on whether the ith symbol is Σ or Π) is this wrong value, Alice must supply a wrong polynomial—that is, a polynomial $A'(x)$ which is different from the correct polynomial, call it $C(x)$, to which this expression evaluates. Now $C(x) - A'(x)$ is a polynomial of degree $2n$, and thus it has at most $2n$ roots (recall Lemma 10.4). Hence, the probability that the random value of x between 0 and $p-1$ generated by Bob will be one of these integers is at most $\frac{2n}{p}$ (this is why we picked p to be at least 2^n). It follows that the probability that the value a at the ith round is correct is at most the probability that the $i-1$st value was correct, times $(1 - \frac{2n}{2^n})$, establishing the claim.

It follows from the claim that in the last round Bob will catch Alice's deception with probability $(1 - \frac{2n}{2^n})^n$, which is arbitrarily close to one for large enough values of n. Finally, to achieve the confidence level of $1 - 2^{-n}$ required by our definition of interactive proofs, it suffices to repeat the protocol twice. □

Corollary: ABPP = IP = PSPACE.

Proof: Just notice that in the interactive protocol in the above proof Alice may very well know the random bits generated by Bob—these random bits are the random evaluation points at each round, a useless information for Alice. Therefore, this protocol can be reformulated as a computation tree for QSAT with nodes alternating between randomizing and maximizing ones, so that both acceptance and rejection require clear majority. □

19.3 MORE PSPACE-COMPLETE PROBLEMS

The following is a basic **PSPACE**-complete problem:

IN-PLACE ACCEPTANCE: Given a deterministic Turing machine M and an input x, does M accept x *without ever leaving the* $|x| + 1$ *first symbols of its string?*

Theorem 19.9: IN-PLACE ACCEPTANCE is **PSPACE**-complete.

Proof: It is obviously in **PSPACE**: In linear space we can simulate M on x, keeping a count of the number of steps. We reject if the machine either rejects, or attempts to add a blank symbol $\sqcup$ (thus violating the "in-place" aspect of the requirement), or operates for more than $|K||x||\Sigma|^{|x|}$ steps.

Suppose then that L is a language in **PSPACE** accepted in space n^k by machine M. Obviously, M accepts x if and only if it accepts *in place* the string $x\sqcup^{n^k}$ (that is, x "padded" with n^k blanks). Hence $x \in L$ if and only if $(M, x\sqcup^{n^k})$ is a "yes" instance of IN-PLACE ACCEPTANCE. □

The following variant is also useful:

IN-PLACE DIVERGENCE: Given the description M of a deterministic Turing machine, does M have a divergent computation that uses at most $|M|$ symbols?

Corollary: IN-PLACE DIVERGENCE is **PSPACE**-complete.

Proof: It is obviously in **PSPACE**; and we can reduce IN-PLACE ACCEPTANCE to IN-PLACE DIVERGENCE as follows: Given M and x we can design a machine M' that starts by writing x on an empty string; M' then simulates M, also keeping a count of the number of steps. When M would be about to accept, M' restores the initial configuration by resetting the count to zero and clearing the string, and therefore diverges. If M exhibits any other behavior on input x (rejection, detection of divergence in terms of an excessive count, attempt to add a blank symbol, etc.) M' halts. It is clear that M' has a divergent computation *starting from any configuration* if and only if it has one from the initial configuration with empty string, which happens if and only if M accepts x. □

There are many problems that can be shown **PSPACE**-complete by reductions from IN-PLACE ACCEPTANCE and IN-PLACE DIVERGENCE. An important category of such problems is concerned with *distributed computation*. We introduce an example next.

A *process* is a directed graph $G = (V, E)$ whose vertices are called *states* and whose edges are called *transitions*. A *system of communicating processes* is a set of processes $\{G_1 = (V_1, E_1), \ldots, G_n = (V_n, E_n)\}$ where all the V_i's are assumed disjoint, together with a set $P = \{\{e_1, e_1'\}, \ldots, \{e_m, e_m'\}\}$ of unordered pairs of transitions called *communication pairs*. Each communication pair $\{e_i, e_i'\} \in P$ is such that $e_i \in E_j$ and $e_i' \in E_k$ for some $j \neq k$. Intuitively, a pair of transitions in P is one way whereby the corresponding processes can communicate, by synchronously changing their state according to the two transitions. For such communication to take place, the two processes must be at the appropriate states (the tails of the two transitions).

The *set of system states* V of a system of communicating processes is the Cartesian product of all V_i's: $V = V_1 \times V_2 \times \ldots \times V_n$. We define the *transition relation* $T \subseteq V \times V$ as follows: $((a_1, \ldots, a_n), (b_1, \ldots, b_n)) \in T$ if and only if there are two indices $j \neq k$ such that $\{(a_j, b_j), (a_k, b_k)\} \in P$, and $a_i = b_i$ for all $i \notin \{j, k\}$. That is, $((a_1, \ldots, a_n), (b_1, \ldots, b_n)) \in T$ if and only we can go from $(a_1, \ldots, a_n)$ to $(b_1, \ldots, b_n)$ by changing two components of the system state according to a pair in P, and keeping all other components the same.

Designing systems of communicating processes so that they conform to various specifications, and testing that the result of the design is correct, are most important problems. Unfortunately, just about all important properties of such systems are intractable to test. For example, let us define a *deadlock system state* to be a system state $d \in V$ such that there is no $a \in V$ with

$(d, a) \in T$. Deadlock system states are undesirable, and it is of interest to detect them. It is not hard to see that the problem of telling whether a given system of communicating processes has a deadlock system state is **NP**-complete (see Problem 19.4.5). However, in actual systems only a small fraction of the system states can be accessed in actual operation; unreachable deadlock states are therefore irrelevant. The following problem is much more interesting:

REACHABLE DEADLOCK: Given a system of communicating processes, and an initial system state a, is there a deadlock system state d *that is reachable from* a *in the transition relation?*

Theorem 19.10: REACHABLE DEADLOCK is **PSPACE**-complete.

Proof: To show that it is in **PSPACE**, suppose that we are given a system of communicating processes. It is easy to see that, using only polynomial working space we can output the whole transition relation. And given the transition relation, in nondeterministic logarithmic space (in the size of the transition relation, which is exponential in the size of the input) we can determine whether any deadlock system state is reachable from a. Finally, using the technique in Proposition 8.2, we can combine the two space-bounded algorithms to a single algorithm that does not have to store the whole transition relation, and thus solves the problem in polynomial space.

To show completeness, we shall reduce IN-PLACE ACCEPTANCE to REACHABLE DEADLOCK. We are given a Turing machine M and an input x, and we are asked whether M accepts x in place. We construct a system of processes as follows: There are $|x|+2$ processes as many squares in the string of M when it computes in place, plus a leading $\triangleright$ and a trailing $\triangleleft$, a new symbol. We assume that we have modified the program of M such that, when the cursor scans a $\triangleleft$ (that is, when the computation is about to violate the in-place restriction), the machine rejects. The $|x| + 2$ processes are all isomorphic: The state space V_i of the ith process is $\{s^i : s \in \Sigma \cup \Sigma \times K\}$, that is, a "marked copy" of the set $\Sigma \cup \Sigma \times K$ this is the set of all symbols that can appear on the computation table of the machine, recall Figure 8.3. Also, $E_i = V_i \times V_i$—that is, all possible edges are present.

We must next define the set P of communication pairs. P contains all pairs of edges of the form $\{(s_1^i, s_2^i), (s_3^{i+1}, s_4^{i+1})\}$ such that, (a) one of the symbols s_1, s_3 and one of the symbols s_2, s_4 is in $\Sigma \times K$, and (b) if at some point two adjacent squares of the string of M contain the symbols s_1 and s_3, in the next step the same squares will contain s_2 and s_4. We also add all pairs of the form $\{(("\text{no}", a)^i, ("\text{no}", a)^i), (b^j, b^j)\}$ for all $a, b \in \Sigma$ and $i, j \leq |x|$ (these will enable rejecting configurations to have a self-transition, thus precluding them from being deadlock system states). Having defined the system of communicating processes, let the initial system state be $a = (\triangleright, (x_1, s), x_2, \ldots, x_n, \triangleleft)$, where $x = x_1 x_2 \ldots x_n$ is the input and s the starting state of M.

It should be clear that configurations of M correspond in a very natural way to system states, and the transition relation corresponds to the configuration graph. Naturally, there are system states that are not configurations (for example, those that do not start with $\triangleright$, or have more than one symbol from $\Sigma \times K$); but such system states will not be reachable from a. Since the machine is deterministic, there is only one path in the transition graph of this process system leaving a, and this is the path through precisely the configurations of the computation of M on input x. If M rejects or diverges, then there is no deadlock reachable from a. If M accepts, then the accepting configuration will be the only reachable deadlock. It follows that there is a reachable deadlock if and only if M accepts x in place. □

Periodic Optimization

Suppose that we must provide machines for tasks that must be executed daily. Each of these tasks starts at the same specific time of each day, ends at another fixed time, and requires the exclusive use of a machine for its duration. For example, in Figure 19.9(a) Task A must be executed between 6 a.m. and 3 p.m. each day, Task B between 1 p.m. and 11 p.m., Task C between 8 p.m. and 8 a.m., and Task D between midnight and 4 a.m. We wish to minimize the number of machines needed (all machines are identical and appropriate for all tasks).

There is a simple graph-theoretic approach to this problem: We consider an undirected graph whose nodes are the tasks. There is an edge between two tasks if the corresponding intervals intersect, and the two tasks must therefore use different machines (see Figure 19.9(b)). It is clear then that the minimum number of machines needed is *the chromatic number of the graph*, that is, the minimum number of colors needed in order to color the nodes of the graph so that no two adjacent nodes have the same color. We conclude that *three* machines are needed in our example.

But it turns out *two machines are enough* in this example! The analysis above is flawed, because it is based upon the *unnecessary restriction that a task uses the same machine every day.* Without this restriction, the graph we need to color is shown in Figure 19.9(c). Here A_i stands for task A at day i; and so on. Despite the fact that this graph is infinite, it is clear that *it can be colored with only two colors.* Notice that this graph is *periodic*, in principle extending to infinity in both directions. Periodic graphs can be described very succinctly as in Figure 19.9(d). Notice that some edges are directed and labeled $+1$; such an edge (u, v) means that there is an undirected edge connecting nodes u_i and v_{i+1} for all integers i. We can now state the problem:

PERIODIC GRAPH COLORING: Given a periodic graph G and a number of colors k, can it be colored with k colors?

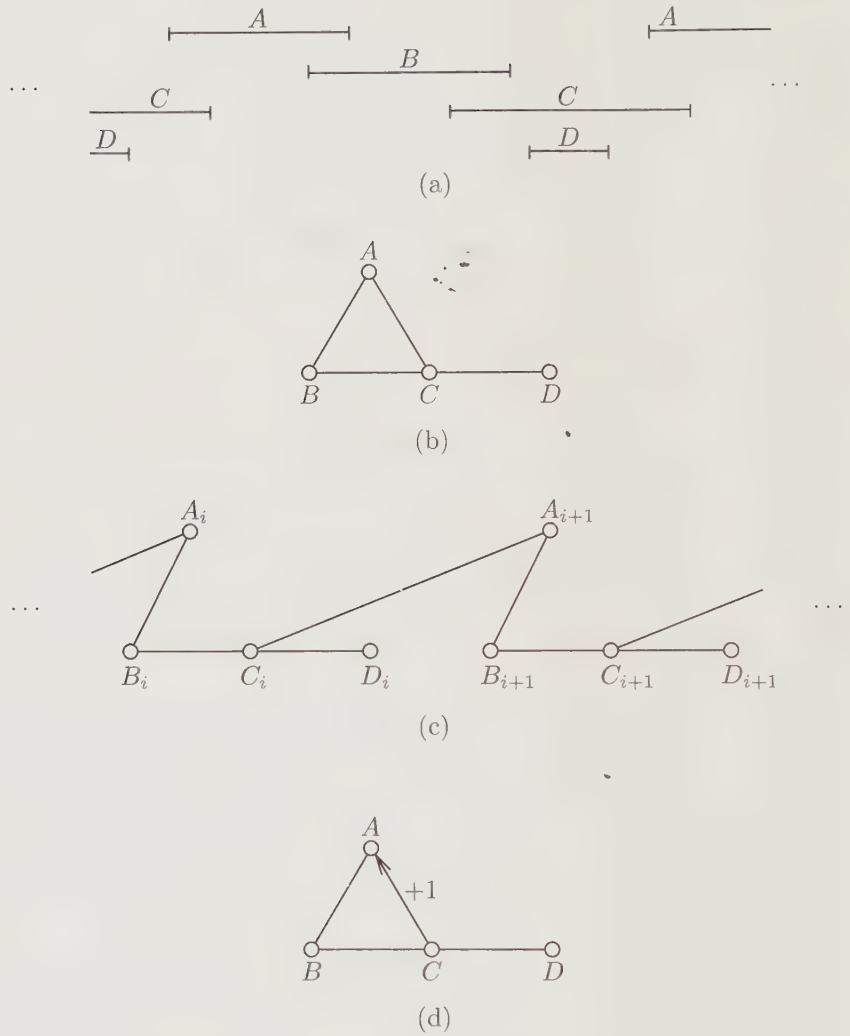


Figure 19-9. Periodic scheduling and graph coloring.

Although coloring an infinite graph may seem unusual and exotic, it is a well-defined problem: Either a coloring (a mapping of the countably infinite set of nodes to $\{1, 2, \ldots, k\}$) exists, or it does not, although the answer is not obviously computable.

Graphs are not the only objects amenable to this generalization. Consider the following periodic Boolean expression in conjunctive normal form:

$$\ldots \wedge (x_i \vee \neg y_{i+1}) \wedge (x_i \vee y_i) \wedge (\neg x_i \vee z_{i+1} \vee y_i) \wedge$$
$$(x_{i+1} \vee \neg y_{i+2}) \wedge (x_{i+1} \vee y_{i+1}) \wedge (\neg x_{i+1} \vee z_{i+2} \vee y_{i+1}) \wedge \ldots$$

It can be rewritten succinctly this way:

$$(x \vee \neg y_{+1}) \wedge (x \vee y) \wedge (\neg x \vee z_{+1} \vee y).$$

For each integer i we have a group of three clauses and three Boolean variables. The subscript in y_{+1} means that in the ith group this clause will contain y_{i+1}, not y_i. Naturally, we could define more complex expressions with clauses that contain variables from more than two blocks; or graphs with edges that connect vertices in non-adjacent blocks (the succinct representations of such objects would have arbitrary positive integers as subscripts). Here we shall only consider periodic objects in which only adjacent blocks interact.

The following problem now suggests itself:

PERIODIC SAT: Given a periodic Boolean expression, is there a truth assignment on its variables that satisfies all clauses?

Again, it is a well-defined question, which is not obviously computable.

Theorem 19.11: PERIODIC SAT is **PSPACE**-complete.

Proof: We first show that the problem is in **PSPACE**. Suppose the given periodic expression is satisfiable, and consider its satisfying truth assignment. This truth assignment consists of a two-way infinite sequence $\ldots, T_i, T_{i+1}, T_{i+2}, \ldots$ of truth assignments to the various blocks of variables. Each T_i is an element of $\{\mathbf{true}, \mathbf{false}\}^n$, where n is the number of variables in each block of the periodic expression. The *ith chunk*, where i is any integer, is the pair (T_i, T_{i+1}) of two consecutive truth assignments. Since there are 2^{2n} possible different chunks, there must be two chunks, not further than 2^{2n} fom each other, that are identical. That is, $(T_i, T_{i+1}) = (T_j, T_{j+1})$ for some i and some j between $i+2$ and $i+2^{(k+1)n}$. But this means that there is a satisfying truth assignment consisting of a two-way infinite repetition of $(T_i, T_{i+1}, \ldots, T_{j-1})$. We conclude that *if a periodic expression is satisfiable, then it has a periodic satisfying truth assignment with period at most exponential* in the number of variables.

This crucial observation allows us to solve the problem in polynomial space: Using nondeterminism we can guess truth assignments $T_1, T_2, \ldots$, always remembering the last two. After we guess T_i we check that all clauses in the $i-1$st group are satisfied. Once we have successfully guessed $T_{2^{2n}+2}$ we accept: We know that there is a periodic satisfying truth assignment.

To show completeness, we shall reduce IN-PLACE DIVERGENCE to PERIODIC SAT. We are given a Turing machine M, and we are asked whether there exists *any* computation of M (from any configuration) that loops forever without leaving the first $|M|$ squares of M's string. Our periodic expression has

enough variables in each block to encode a configuration of M (plus some additional variables). Its clauses express the requirement that the $i+1$st block of variables encodes a configuration that follows in one step of M from the configuration encoded by the ith block. This is done in very much the same way as in Cook's Theorem (Theorem 8.2): A circuit can capture the transition relation of M; the gates of the circuit are the new variables, and the clauses state that the inputs and output of each gate stand in their proper logical relationship.

It follows that if the periodic expression thus constructed is satisfiable, then the truth values of the variables determine an infinite computation of M on its first $|M|$ squares. Conversely, if an infinite, in-place computation of M exists, then there is an infinite computation that consists of the infinite repetition of a finite "loop" (in proof, just run the infinite computation long enough until a configuration repeats). And such infinite computation yields a two-way infinite truth assignment that satisfies the constructed expression. □

Theorem 19.12: PERIODIC GRAPH COLORING is **PSPACE**-complete.

Proof: The proof that it is in **PSPACE** is the same as for PERIODIC SAT: The coloring of two adjacent graphs must repeat after at most k^n copies, and thus the whole coloring can be assumed to be periodic with exponential period. A polynomial-space machine can then guess and check such a coloring.

To show completeness, we mimic the reduction from 3SAT to COLORING in Chapter 9. First, recall our proof in Theorem 9.3 that the variant NOT-ALL-EQUAL SAT of satisfiability is **NP**-complete; the observation there is that the clauses produced from circuits are satisfiable in the ordinary sense if and only if they are satisfiable in the not-all-equal sense. It follows that the proof of Theorem 19.11 above also establishes that the PERIODIC NOT-ALL-EQUAL SAT problem (obvious definition) is **PSPACE**-complete.

Finally, recall the reduction from NOT-ALL-EQUAL SAT to COLORING (Figure 9.8). To adapt this reduction to the periodic versions of the two problems, create a new set of triangles meeting at a different 2-node for each integer i, as well as a new set of node-disjoint triangles for the clauses at the ith level. The nodes of the clause triangles are adjacent to the literals appearing in the clause (which may include some literals at the next level). We make sure that all 2-nodes are colored the same color by adding at each level i two new adjacent nodes, call them a_i and b_i, and connecting them to the 2-nodes at both the i and $(i+1)$st level. This completes our proof that PERIODIC COLORING is **PSPACE**-complete. □

There are many more **PSPACE**-complete problems of this type; see the references in 19.4.6.

19.4 NOTES, REFERENCES, AND PROBLEMS

19.4.1 The **PSPACE**-completeness of QSAT was established in

- L. J. Stockmeyer and A. R. Meyer "Word problems requiring exponential time," *Proc. 5th ACM Symp. on the Theory of Computing*, pp. 1–9, 1973.

For another important **PSPACE**-complete problem from the same paper, *regular expression equivalence*, see Problem 20.2.13. Karp's paper

- R. M. Karp "Reducibility among combinatorial problems," pp. 85–103 in Complexity of Computer Computations, edited by J. W. Thatcher and R. E. Miller, Plenum Press, New York, 1972.

also contains a **PSPACE**-complete problem: Context sensitive recognition (given a context-sensitive grammar (recall Problem 3.4.2) and a string, is the string in the language generated by the grammar?). This problem is closely related to our IN-PLACE ACCEPTANCE (Theorem 19.9).

The game-theoretic aspect of **PSPACE** was pointed out in

- T. J. Schäfer "Complexity of some two-person perfect-information games," *J.CSS 16*, pp. 185–225, 1978,

where Theorem 19.3 on GEOGRAPHY was proven. The result on GO is from

- D. Lichtenstein, M. Sipser "GO is polynomial-space hard," *J.ACM, 27*, pp. 393–401, 1980.

A generalization of the game of HEX (or "blockers") to an arbitrary graph is also **PSPACE**-complete:

- S. Even and R. E. Tarjan "A combinatorial game which is complete for polynomial space," *J.ACM 23*, pp. 710–719, 1976,

and so is the game of checkers:

- A. S. Fraenkel, M. R. Garey, D. S. Johnson, T. Schäfer, and Y. Yesha "The complexity of checkers on an $N \times N$ board –preliminary report," *Proc. 19th IEEE Symp. on the Foundations of Computer Science*, pp. 55–64, 1978.

Generalized chess is even harder, essentially because chess termination rules allow for exponentially long games (we have artificially removed this possibility in GO):

- A. S. Fraenkel and D. Lichtenstein "Computing a perfect strategy for $n \times n$ chess requires time exponential in n," *J. Combin. Theory Series A, 31*, pp. 199–213, 1981.

19.4.2 But even *solitaire* games can be **PSPACE**-complete. Let G be a directed acyclic graph. We wish to place a *pebble* on every node of the graph. A pebble can be placed on a node if all predecessors of the node currently have pebbles. We can place a pebble on a source any time—that's how the game starts –and we can remove a pebble from any node anytime. The game ends when all nodes have had a pebble

placed on them at some point. *We wish to minimize the necessary pebble supply,* that is, the maximum number of pebbles that have to be simultaneously present on the graph at any time. The difficulty that makes this problem harder than **NP**-complete is, of course, that nodes may have to be *repebbled* many times, in order to achieve the minimum number of pebbles. It has been shown that this problem, which is related to *register minimization* and Problem 7.4.17, is **PSPACE**-complete:

- J. R. Gilbert, T. Lengauer, and R. E. Tarjan "The pebbling problem is complete for polymomial space," *SIAM J. Comp. 9*, pp. 513–524, 1980.

For another **PSPACE**-complete solitaire game, consider the problem of moving a complicated piece of furniture, with parts and attachments that can move and rotate, through an oddly shaped corridor; see

- J. Reif "Complexity of the mover's problem and generalizations," *Proc. 20th IEEE Symp. on the Foundations of Computer Science*, pp. 144–154, 1979; see also
- J. E. Hopcroft, J. T. Schwartz, and M. Sharir "On the complexity of motion planning for multiple independent objects: PSPACE-completeness of the warehouseman's problem," *Int. J. Robotics Research, 3*, pp. 76–88, 1984

for a related problem.

19.4.3 The complexity of problems of decision-making under uncertainty such as STOCHASTIC SCHEDULING were studied in

- C. H. Papadimitriou "Games against nature," *Proc. 24th IEEE Symp. on the Foundations of Computer Science*, pp. 446–450; also *J.CSS 31*, pp. 288–301, 1985,

where several other examples of complete problems can be found, and the equality of this class with **PSPACE** was proved (Theorem 19.5).

19.4.4 Shamir's theorem, from

- A. Shamir "IP=PSPACE," *Proc. 31st IEEE Symp. on the Foundations of Computer Science*, pp. , 11–15, 1990,

improves upon a result that predated it by a few days, and states that **IP** contains **PH**, the polynomial hierarchy:

- C. Lund, L. Fortnow, H. Karloff, and N. Nisan "Algebraic methods for interactive proofs," *Proc. 31st IEEE Symp. on the Foundations of Computer Science*, pp. , 1–10, 1990.

The latter proof used arithmetic interactive techniques very much like Shamir's to evaluate *the permanent*, relying on Toda's theorem (recall Section 18.2). The ideas in these arithmetic techniques can be traced to research related to aspects of cryptography and testing:

- D. Beaver and J. Feigenbaum "Hiding instances in multioracle queries," *Proc. 7th Symp. on Theoretical Aspects of Comp. Science*, Lecture Notes in Computer Science, Springer Verlag, Berlin, pp. 37–48, 1990, and

- R. J. Lipton "New directions in testing," pp. 191–202 in *Distributed Computing and Cryptography*, American Math. Society, Providence, 1991.

For the research activity that followed Shamir's theorem on *multi-prover protocols*, and culminated in a breakthrough in yet another unexpected direction, see the references in 20.2.16.

19.4.5 Problem: Show that the problem of telling whether a given system of communicating processes has a deadlock state (reachable or not) is **NP**-complete.

19.4.6 Theorems 19.11 and 19.12 on PERIODIC SAT and COLORING are only a few of the possibilities:

Problem: Define *carefully* the PERIODIC HAMILTON PATH and the PERIODIC INDEPENDENT SET problem, and show they are **PSPACE**-complete.

For more on this subject see

- J. B. Orlin "The complexity of dynamic languages and dynamic optimization problems," *Proc. 13th ACM Symp. on the Theory of Computing*, pp. 218–227, 1981.

CHAPTER

20 A GLIMPSE BEYOND

In this chapter we shall see at last some truly, provably intractable problems...

20.1 EXPONENTIAL TIME

Recall our definition of exponential time

$$\mathbf{EXP} = \mathbf{TIME}(2^{n^k}),$$

and the corresponding nondeterministic class

$$\mathbf{NEXP} = \mathbf{NTIME}(2^{n^k}).$$

The counterpart of $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ at this level is the $\mathbf{EXP} \stackrel{?}{=} \mathbf{NEXP}$ question — unfortunately, we are not any closer to resolving it. However, there is something simple that can be said about the relation between these two problems:

Theorem 20.1: If $\mathbf{P} = \mathbf{NP}$ then $\mathbf{EXP} = \mathbf{NEXP}$.

Proof: Let $L \in \mathbf{NEXP}$; under the assumption that $\mathbf{P} = \mathbf{NP}$, we shall show that it is in **EXP**. By definition, L is decided by a nondeterministic Turing machine N in time 2^{n^k}, for some k. Consider now the "exponentially padded version" of L:

$$L' = \{x\sqcap^{2^{|x|^k}-|x|} : x \in L\}.$$

That is, L' consists of all strings x in L padded by enough "quasiblanks" to bring the total length of the string to $2^{|x|^k}$.

We claim that $L' \in \mathbf{NP}$. This is easy to show: The polynomial-time nondeterministic machine that decides L' is precisely N, slightly modified so

that it first checks whether its string ends in exponentially many quasiblanks; if not, it rejects, otherwise it simulates N, treating quasiblanks as blanks. The machine works in polynomial time simply because its input is exponentially long.

Since $L' \in \mathbf{NP}$, and we are assuming that $\mathbf{P} = \mathbf{NP}$, we know that $L' \in \mathbf{P}$. Thus there is a deterministic Turing machine M' that decides L' in time n^ℓ, say. We can in fact assume that M' is a machine with input and output, so that it never writes on its input string. *We shall now invert the previous construction* to get a deterministic machine M that decides L in time 2^{n^ℓ}, for some integer ℓ, thus completing the proof. But this is easy to do: M on input x simply simulates M' on input $x\sqcap^{2^{|x|^k}-|x|}$. The only difficulty, tracking down M''s input cursor when it wanders off in the $\sqcap$'s, can be solved by maintaining the position of the input cursor as an integer in binary. □

Contrapositively, if $\mathbf{EXP} \neq \mathbf{NEXP}$ then $\mathbf{P} \neq \mathbf{NP}$. That is, *class equality propagates upwards*, while *class inequality propagates downwards*. In other words, showing that $\mathbf{EXP} \neq \mathbf{NEXP}$, as we fully believe is the case, might turn out to be even harder than showing $\mathbf{P} \neq \mathbf{NP}$; conceivably $\mathbf{P} \neq \mathbf{NP}$ and still $\mathbf{EXP} = \mathbf{NEXP}$ (see the references).

This can be generalized, of course (for a proof, see Problem 20.2.3):

Corollary: If $f(n)$ and $g(n) \geq n$ are proper functions, then $\mathbf{TIME}(f(n)) = \mathbf{NTIME}(f(n))$ implies $\mathbf{TIME}(g(f(n))) = \mathbf{NTIME}(g(f(n)))$. □

Also, for the analog of Theorem 20.1 in space complexity, as well as for the interaction between time and space complexity see Problem 20.2.4.

It is interesting to compare **EXP** and **NEXP** with two other classes that capture a more benign aspect of exponential time:

$$\mathbf{E} = \mathbf{TIME}(k^n), \text{ and } \mathbf{NE} = \mathbf{NTIME}(k^n).$$

That is, the time bounds in these classes have *linear*, not polynomial, exponent. The main drawback of these classes is that they are not closed under reductions (recall Problem 7.4.4). Still, they are closely related to **EXP** and **NEXP**:

Lemma 20.1: For any language L in **NEXP** there is a language L' in **NE** such that L reduces to L'.

Proof: : Just notice that, if $L \in \mathbf{TIME}(2^{n^k})$, then $L' = \{x\sqcap^{|x|^k} : x \in L\}$ is in **NE**, and L reduces to it. □

To put it otherwise, **NEXP** is the closure under reductions of **NE**. This lemma is useful in proving **NEXP**-completeness results.

Succinct Problems

But what kinds of complete problems do these classes have? In the next subsection we shall see some interesting problems from logic that capture this level of

complexity. Another interesting kind of **EXP**- and **NEXP**-complete problems comes from the consideration that led to Theorem 20.1: **NEXP** and **EXP** are nothing else but **P** and **NP** *on exponentially more succinct input.*

Several **NP**-complete graph-theoretic problems (including MAX CUT, MAX FLOW, BISECTION WIDTH, and so on) have important applications to the automated design of VLSI chips. In chip design, however, there are ways of describing chips that are not explicit and direct, listing all components and connections of the chip, but *succinct* and *implicit*, describing a chip in terms of repeated patterns and encoded configurations. For a simple example, a highly regular circuit could be described like this:

"Repeat the (given) component C horizontally and vertically in an $N \times M$ grid $(i, j), i = 1, \ldots, N; j = 1, \ldots, M$ (where N and M are given large integers), except at the positions $i = j$, the positions $i = 2$, and the positions $i = N - 1$; at these positions place the other given component $C' \ldots$"

And so on. Since M and N are given in binary, such descriptions are conceivably *exponentially more succinct* than the circuits they describe. Accordingly, the graphs that abstract the structure and operation of such circuits (and for which we need to solve several computational problems, such as MAX-CUT, BISECTION WIDTH, and so on) are described in a manner exponentially more succinct than our usual explicit representation that lists all edges.

We can define a way of representing graphs that captures the effect of such "hardware description languages." A *succinct representation of a graph with n nodes*, where $n = 2^b$ is a power of two, is a Boolean circuit C with $2b$ input gates. The graph represented by C, denoted G_C, is defined as follows: The nodes of G_C are $\{1, 2, \ldots, n\}$. And $[i, j]$ is an edge of G_C *if and only if* C *accepts the binary representations of the b-bit integers i, j as inputs.*

The problem SUCCINCT HAMILTON PATH is now this: Given the succinct representation C of a graph G_C with n nodes, does G_C have a Hamilton path? Similarly for SUCCINCT MAX CUT, SUCCINCT BISECTION WIDTH, or the succinct version of any graph-theoretic problem (for these latter problems, as well as for any graph-theoretic optimization problem, a binary integer budget/goal K is provided along with C).

We can also define SUCCINCT 3SAT, SUCCINCT CIRCUIT SAT, and SUCCINCT CIRCUIT VALUE. To encode Boolean circuits, we first assume that all gates are predecessors to at most two other gates. That is, we think that a gate in the circuit has four *neighbors*, of which the first two are predecessors and the other two successors (if there are fewer neighbors, the missing neighbors are set to a fictitious gate 0, say). The succinct representation of a Boolean circuit is another Boolean circuit with many outputs. On input of the form $i; k$, where i is a gate number in binary and $0 \leq k \leq 3$, the output of the encoding circuit is of the form $j; s$, where gate j is the kth neighbor of gate i, and s encodes the

sort (AND, OR, NOT) of gate i. To encode succinctly a Boolean expression in conjunctive normal form, we assume that all clauses have three literals and each literal appears three times (again, with missing literals and clauses represented as 0). Suppose that the encoded expression has n variables and m clauses. The encoding circuit on input $0; i; k$, where $i \leq n$ and $k \leq 2$, returns the index of the clause where the literal $\neg x_i$ appears for the kth time; on input $1; i; k$ it returns the number of the clause where the literal x_i appears for the kth time; and on input $2; i; k$, with $i \leq m$ and $1 \leq k \leq 3$, it returns the kth literal of clause i. It should be clear that all Boolean expressions can be thus encoded. SUCCINCT 3SAT is now the problem of telling, given a circuit C, whether the Boolean expression ϕ_C represented by it[†] is satisfiable. Similarly for SUCCINCT CIRCUIT SAT and and SUCCINCT CIRCUIT VALUE.

Theorem 20.2: SUCCINCT CIRCUIT SAT is **NEXP**-complete.

Proof: It is clear that the problem is in **NEXP**: A nondeterministic machine can guess a satisfying truth assignment for all gates $[t_1, \ldots, t_N]$, where N is exponential in the size of the input C, and then verify that the output gate is **true** and all gates have legitimate values.

To prove completeness, we shall reduce any language in **NEXP** to SUCCINCT CIRCUIT SAT. So, suppose that L is a language decided by a nondeterministic Turing machine N in time 2^n (here we use Lemma 20.1). For each input x we shall construct an instance $R(x)$ of SUCCINCT CIRCUIT SAT. $R(x)$ is a circuit, which encodes another circuit $C_{R(x)}$, with the following property: $C_{R(x)}$ is satisfiable if and only if $x \in L$. The circuit $C_{R(x)}$ is essentially the one constructed in the proof of Cook's theorem (Theorem 8.2), only exponentially larger. That is, there are now going to be $2^n \times 2^n$ copies of the basic circuit C. The gates of $C_{R(x)}$ will thus be of the form i, j, k, where $i, j \leq 2^n$, and $k \leq |C|$, where C is the size of the basic circuit. $R(x)$ on input i, j, k outputs in binary $s; i, j, k'; i, j, k''$, where s is an encoding of the sort of gate k of C, and k', k'' are the predecessors of k in C. It is very easy to finish the construction of $R(x)$ so that it appropriately identifies inputs and outputs of adjacent copies of C, and forces the upper and lower row and leftmost and rightmost columns of the computation table to contain the correct symbols (recall Figure 8.3). □

For more discussion of the complexity of problems of this sort see Problem 20.2.9. A proof very similar to that of Theorem 20.2 above establishes the following:

Corollary 1: The problems SUCCINCT 3SAT and SUCCINCT HAMILTON PATH are **NEXP**-complete.

Proof: It is clear that SUCCINCT CIRCUIT SAT and SUCCINCT 3SAT are in

[†] *If* such an expression exists, that is. Most circuits fail to encode a legitimate Boolean expression for any one of a long array of possible reasons.

NEXP. For completeness, the ordinary reduction from CIRCUIT SAT to 3SAT (recall Example 8.3) can be modified in a straight-forward way to establish that SUCCINCT CIRCUIT SAT is reducible to SUCCINCT 3SAT. Given a circuit K encoding some circuit, call it C_K, we must construct a circuit $R(K)$ which encodes an equivalent expression $\phi_{R(K)}$. Expression $\phi_{R(K)}$ must have as many variables as C_K has gates, and twice as many clauses, with a structure that directly reflects that of C_K. This is quite easy to do; $R(K)$ is essentially K with some simple pre-processing of the input and post-processing of the output to conform with the new conventions.

We shall now reduce SUCCINCT 3SAT to SUCCINCT HAMILTON PATH. Given a circuit C describing a Boolean expression ϕ_C, we can construct the circuit $R(C)$ which encodes the graph resulting from our reduction from 3SAT to HAMILTON PATH (Theorem 9.7). The graph will have three nodes for each variable corresponding to the choice gadget, recall Figure 9.7), three for each clause (the nodes of the triangle in the constraint gadget, Figure 9.7), plus twelve nodes for each occurrence of a literal to a clause (the exclusive-or gadget). Whether any two such nodes are connected by an edge in the graph can be easily determined from the indices of the two nodes, plus the occurrence relation of the Boolean expression, as described by circuit C. Therefore, the circuit $R(C)$ that encodes the resulting graph can be again designed so that it is C with some easy pre-processing and post-processing of indices. □

Corollary 2: SUCCINCT CIRCUIT VALUE is **EXP**-complete.

Proof: It is clear that it is in **EXP**. The deterministic version of the proof of Theorem 20.2 (recall Theorems 8.1 and 8.2) establishes completeness. □

Finally we have the exponential counterpart of Corollary 1 of Theorem 16.5 (which could also be obtained by a "padding argument" like the one used in the proof of Theorem 20.1):

Corollary 3: **EXP** coincides with alternating polynomial space.

Proof: SUCCINCT CIRCUIT VALUE is complete for both classes (see Problem 20.2.9(e)). □

SUCCINCT 3SAT plays a central role in the study of interactive protocols, and ultimately of approximability, see 13.4.14 and thereafter.

A Special Case of First-Order Logic

FIRST-ORDER SAT, the problem of telling whether an expression in first-order logic has a model, is of course undecidable (Corollary 1 to Theorem 6.3). However, there are several interesting "syntactic classes" of expressions for which this problem is decidable. We shall examine one of them below (see the references in 20.2.11 for many others).

An expression in first-order logic is said to be a *Schönfinkel-Bernays* expression if (a) its alphabet has only relational and constant symbols, no function symbols, and *no equality;* and (b) it is of the form

$$\psi = \underline{\exists x_1 \ldots \exists x_k \forall y_1 \ldots \forall y_\ell \phi}, \tag{1}$$

that is, it is in prenex form with a sequence of existential quantifiers followed by a sequence of universal ones. SCHÖNFINKEL-BERNAYS SAT is this problem: Given a Schönfinkel-Bernays expression as in equation (1), does it have a model?

Theorem 20.3: SCHÖNFINKEL-BERNAYS SAT is **NEXP**-complete.

Proof: We shall first show that it is in **NEXP**. Consider a Schönfinkel-Bernays expression ψ as in (1), and suppose that there are m constants appearing in ϕ.

Lemma 20.2: ψ is satisfiable if and only if it has a model with $k+m$ or fewer elements.

Proof: Suppose that ψ has a model M with a universe U. By the definition of satisfaction, there are elements $u_1, \ldots, u_k \in U$, not necessarily distinct, such that $M_{x_1=u_1,\ldots,x_k=u_k} \models \underline{\forall y_1 \ldots \forall y_\ell \phi}$. Now let U' be the set $\{u_1 \ldots, u_k\}$, plus all elements of U that are images under M of some constant appearing in ϕ (notice that the set U' has at most $k+m$ elements); and let M' be M *restricted to* U'. That is, M' has universe U', and maps all constant symbols in the vocabulary to the same elements of U' as M (notice that the images of constants are by definition in U'). Finally, a k-tuple of elements of U' is related by relation symbol R under M' if and only if it is under M.

We claim that $M' \models \psi$. The reason is that, since $M_{x_1=u_1,\ldots,x_k=u_k} \models \underline{\forall y_1 \ldots \forall y_\ell \phi}$, then certainly $M'_{x_1=u_1,\ldots,x_k=u_k} \models \underline{\forall y_1 \ldots \forall y_\ell \phi}$, because M and M' agree on all constant and relation symbols, and deleting elements from the universe makes a universal sentence even easier to satisfy. □

That SCHÖNFINKEL-BERNAYS SAT is in **NEXP** follows immediately from this lemma: To verify that an Schönfinkel-Bernays expression is satisfiable, all we have to do is guess a model with $|U| \leq k+m$ elements, and verify that this model satisfies ψ. Let n be the length of the representation of expression ψ. Since $k+m \leq n$, and each relation and function symbol appearing in ψ has arity at most n, the length of the description of the model is $\mathcal{O}(n^{2n})$; and testing satisfiability can be done in time $\mathcal{O}(n^q)$, where q is the total number of quantifiers. We conclude that the problem is in **NEXP**.

To show completeness, consider a language L decided by a nondeterministic Turing machine N, with two nondeterministic choices per step, in time 2^n (using Lemma 20.1). For each input x we shall construct in logarithmic space a Schönfinkel-Bernays expression $R(x)$ such that $x \in L$ if and only if $R(x)$ has a model.

The construction is essentially the same as that in the proof of Fagin's theorem (Theorem 8.3), except in several ways simpler. Now we do not need

second-order existential quantifiers, because asking whether a model exists has the same effect. The arity of the relation symbols can now depend on n, and this simplifies matters tremendously.

We have $2n$ variables $x_1, \ldots, x_n, y_1, \ldots, y_n$. The whole expression $R(x)$ consists of the conjunction of all expressions described below, preceded by the universal quantification of all $2n$ variables. To simulate the values 0 and 1 for the variables, we have a *unary relation symbol* called $\mathbf{1}(\cdot)$. Intuitively, $\mathbf{1}(x_1)$ means that $x_1 = 1$, and $\neg\mathbf{1}(x_1)$ means that $x_1 = 0$ (we need this trick because we have no equality in our language).

For $k = 1, \ldots, n$ we have a $2k$-ary predicate $S_k(x_1, \ldots, x_k, y_1, \ldots, y_k)$ expressing that the y_i's spell in binary the successor of the number spelled in binary by the x_i's. We can define S_k inductively exactly as in the proof of Theorem 8.3; and for $k = 1$, we have $S_1(x_1, y_1) \Leftrightarrow (\neg\mathbf{1}(x_1) \wedge \mathbf{1}(y_1))$.

For each symbol σ that can appear on the computation table of N we have a $2n$-ary relation symbol $T_\sigma(\mathbf{x}, \mathbf{y})$ (where $\mathbf{x}$ stands for $x_1, \ldots, x_n$ and $\mathbf{y}$ stands for $y_1, \ldots, y_n$), expressing that at the $\mathbf{x}$th step (where $\mathbf{x}$ is interpreted as a n-bit binary integer) the $\mathbf{y}$th symbol of N's string is a σ. There are two n-ary relations C_0 and C_1, where $C_0(\mathbf{x})$ means that at the $\mathbf{x}$th step nondeterministic choice 0 was taken, and $C_1(\mathbf{x})$ means that choice 1 was followed. Again we require that at each step exactly one of the two happens. We also require that the first row of the table is filled properly by x followed by blanks, and that the leftmost and rightmost columns contain only $\triangleright$'s and $\sqcup$'s, respectively. For each quintuple $(\alpha, \beta, \gamma, c, \sigma)$ such that, if three consecutive string symbols are $\alpha\beta\gamma$ and choice c is made, then σ appears in the next step in the place of β, we have an expression that guarantees this, exactly as in the proof of Theorem 8.3. Finally, we require that there be a "yes" in the last row. This concludes our sketch of the construction of $R(x)$. It is easy to argue that the conjuncts of $R(x)$ completely axiomatize the intended meaning of the relation symbols, and thus there is a model for $R(x)$ if and only if $x \in L$. □

One last reminder about **EXP** and **NEXP**-complete problems: Unlike all other completeness results in previous chapters in this book, these are problems that *we know* are not in **P**, and therefore *provably intractable* according to our criterion.

And Beyond...

There is no reason to stop at **NEXP**: Beyond that one finds of course the *exponential hierarchy*, with its alternations of quantifiers. It is supposed that, as in the polynomial case, this hierarchy is indeed an infinite increasing sequence of classes; however, the same hierarchy starting from the class **NE** *does collapse* (see the references). Then we have exponential space **EXPSPACE** = **SPACE**(2^{n^k}); and even further we arrive at *doubly exponential time:* 2-**EXP** =

$\mathbf{TIME}(2^{2^{n^k}})$. Of course there is also 2-$\mathbf{NEXP} = \mathbf{NTIME}(2^{2^{n^k}})$. If 2-$\mathbf{EXP}$ and 2-$\mathbf{NEXP}$ were equal, equality would propagate down to $\mathbf{EXP} = \mathbf{NEXP}$ and finally $\mathbf{P} = \mathbf{NP}$. Further up is the class 3-$\mathbf{EXP} = \mathbf{TIME}(2^{2^{2^{n^k}}})$, and so on.

We have thus an *exponential hierarchy*—for a change, this time a true, provable hierarchy, since each of these classes properly includes the previous one by the time hierarchy theorem. The cumulative complexity class of this hierarchy is called *the class of elementary*† *languages*. That is, a language is elementary if it is in the class

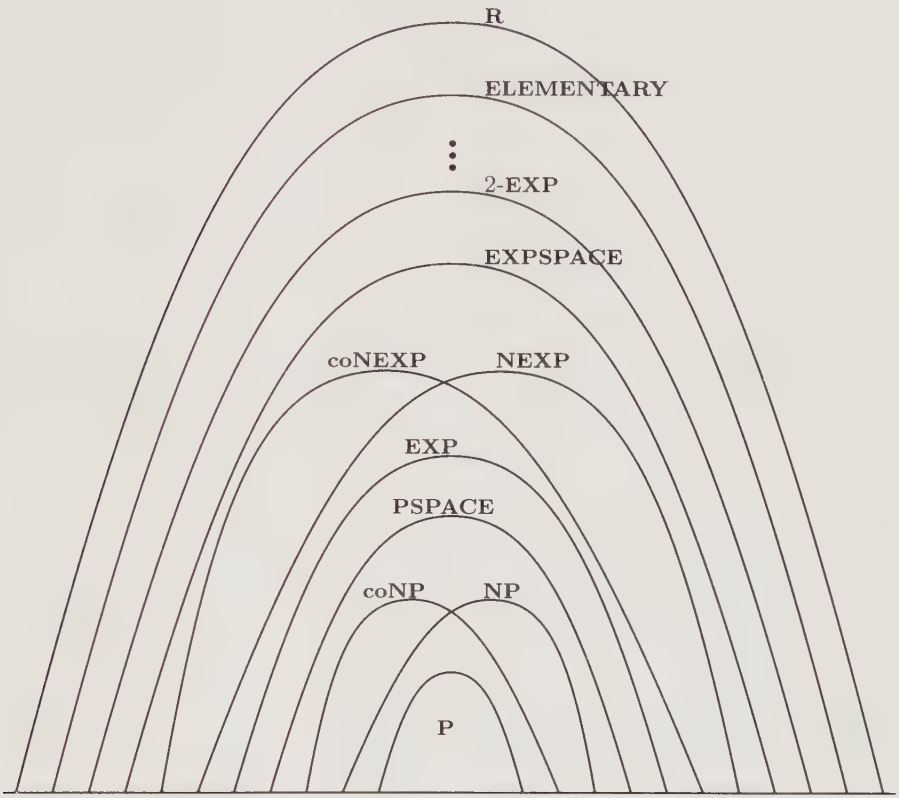
$$\mathbf{TIME}(2^{2^{\cdot^{\cdot^{\cdot^{2^n}}}}}),$$

for some finite number of exponents. Notice that, in this context, nondeterminism, alternation, space bounds, or the n^k in the final exponent are insignificant details... As it turns out, there are some fairly natural, decidable problems that are not even elementary (see Problem 20.2.13).

† The optimism in this term may seem a little overstated; the term was introduced in the context of undecidability.

20.2 NOTES, REFERENCES, AND PROBLEMS

20.2.1 Class review:



There is much confusion in the literature regarding the notation for exponential complexity classes. For example, **EXPTIME** has been sometimes used to denote our **E**, and similarly for nondeterministic classes.

20.2.2 Oracles are known for which $\mathbf{P} \neq \mathbf{NP}$, and still $\mathbf{EXP} = \mathbf{NEXP}$, see

- M. I. Dekhtyar "On the relativization of deterministic and nondeterministic complexity classes," in *Mathematical Foundations of Computer Science*, pp. 255-259, Lecture Notes in Computer Science, vol. 45, Springer Verlag, Berlin, 1976.

Also, oracles are given in this paper that separate **EXP** from **PSPACE**. The relation of these classes with **E** and **NE** is also subject to all kinds of relativizations.

20.2.3 Problem: Show that, if $f(n)$ and $g(n) \geq n$ are proper complexity functions,
(a) $\mathbf{TIME}(f(n)) = \mathbf{NTIME}(f(n))$ implies $\mathbf{TIME}(g(f(n))) = \mathbf{NTIME}(g(f(n)))$;
(b) Similarly for space.

20.2.4 Problem: Show that, if $\mathbf{L} = \mathbf{P}$ then $\mathbf{PSPACE} = \mathbf{EXP}$.

20.2.5 Problem (the nondeterministic space hierarchy): (a) Show that $\mathbf{NSPACE}(n^3) \neq \mathbf{NSPACE}(n^4)$. (Use Problem 20.2.3(b) repeatedly, combined with Savitch's theorem and the space hierarchy theorem.)

(b) More generally, show that $\mathbf{NSPACE}(f(n)) \neq \mathbf{NSPACE}(f^{1+\epsilon}(n))$ for any proper function $f \geq \log n$ and $\epsilon > 0$.

20.2.6 Theorem 20.1, as well as Problems 7.4.7 and 20.2.5, are from

- R. V. Book "Comparing complexity classes," *J.CSS, 9*, pp. 213–229, 1974, and
- R. V. Book "Translational lemmas, polynomial time, and $\log^j n$ space," *Theoretical Comp. Science 1*, pp. 215–226, 1976.

20.2.7 Problem: Show that $\mathbf{P}^{\mathbf{E}} = \mathbf{EXP}$.

20.2.8 Problem: (a) Show that $\mathbf{E} \neq \mathbf{NE}$ if and only if there are unary languages in $\mathbf{NP} - \mathbf{P}$. (Consider the unary version of any language in **E** (or **NE**); show it is in **P** (respectively, **NP**).) This result can be strengthened as follows:

(b) Show that $\mathbf{E} \neq \mathbf{NE}$ if and only if there are sparse languages in $\mathbf{NP} - \mathbf{P}$. (This is from

- J. Hartmanis, V. Sewelson, and N. Immerman "Sparse sets in NP−P: EXPTIME vs. NEXPTIME," *Information and Control, 65*, pp. 158–181, 1985.)

That the **NE** hierarchy collapses was shown in

- L. Hemachandra "The strong exponential hierarchy collapses" *J.CSS 39*, 3, pp. 299–322, 1989.

20.2.9 Succinctness tends to increase the complexity of a problem by an exponential; Theorem 20.2 is only one example of the possibilities:

(a) Define SUCCINCT KNAPSACK and show that it is **NEXP**-complete.

(b) Show that SUCCINCT REACHABILITY is **PSPACE**-complete. Repeat for the case in which the graph is known to be a tree (recall Problem 16.4.4).

(c) Define SUCCINCT ODD MAX FLOW and show that it is **EXP**-complete.

(d) Define NON-EMPTINESS to be the following problem: Given a graph, does it have an edge? Show that SUCCINCT NON-EMPTINESS is **NP**-complete.

(e) Show that SUCCINCT CIRCUIT VALUE is complete for alternating polynomial space.

The complexity of succinct versions of graph-theoretic problems was studied in

- H. Galperin and A. Wigderson "Succinct representations of graphs," *Information and Control, 56*, pp. 183–198, 1983,

where the exponential increase in the complexity of these problems was first observed. The general reduction technique in the proof of Theorem 20.2 is from

- C. H. Papadimitriou and M. Yannakakis "A note on succinct representations of graphs," *Information and Control, 71*, pp. 181–185, 1986.

For a much more detailed treatment of the subject see

- J. L. Balcázar, A. Lozano, and J. Torán "The complexity of algorithmic problems in succinct instances," in *Computer Science,* edited by R. Baeza-Yates and U. Manber, Plenum, New York, 1992.

20.2.10 Problem: We are given a set of square tile types $T = \{t_0, \ldots, t_k\}$, together with two relations $H, V \subseteq T \times T$ (the *horizontal* and *vertical compatibility relations*, respectively). We are also given an integer n in binary. An $n \times n$ *tiling* is a function $f : \{1, \ldots, n\} \times \{1, \ldots, n\} \mapsto T$ such that (a) $f(1,1) = t_0$, and (b) for all i, j $(f(i,j), f(i+1,j)) \in H$, and $(f(i,j), f(i,j+1)) \in V$. TILING is the problem of telling, given T, H, V, and n, whether an $n \times n$ tiling exists.

(a) Show that TILING is **NEXP**-complete.

(b) Show that TILING becomes **NP**-complete if n is given in unary (this is the succinctness phenomenon, backwards).

(c) Show that it is undecidable to tell, given T, H, and V, whether an $n \times n$ tiling exists *for all* $n > 0$.

20.2.11 Decidable fragments of first-order logic. Besides the Schönfinkel-Bernays fragment of first-order logic shown **NEXP**-complete in Theorem 20.3, satisfiability can also be decided for function-free expressions with the following kinds of quantifier sequences:

(a) Quantifiers of the form $\exists^*\forall\exists^*$ (that is, only one universal quantifier). This is the *Ackermann class*, and is **EXP**-complete.

(b) Quantifiers of the form $\exists^*\forall\forall\exists^*$ (that is, only two consecutive universal quantifiers). This is the *Gödel class*, and its satisfiability problem is **NEXP**-complete.

As it turns out, *the validity problem for all other quantifier sequences is undecidable.* Yet another decidable case is this:

(c) Arbitrary expressions in vocabularies with *only unary relation symbols.* This is the *monadic* case, and its satisfiability problem is **NEXP**-complete.

Research on these decidability results was a major part of Hilbert's program (see the references in Chapter 6), and generally predated the undecidability of first-order logic. For decidability, undecidability, and complexity results concerning segments of first-order logic see, respectively,

- B. S. Dreben and W. D. Goldfarb *The Decision Problem: Solvable Cases of Quantification Formulas,* Addison-Wesley, Reading, Massachusetts, 1979.

- H. R. Lewis *Unsolvable Classes of Quantification Formulas*, Addison-Wesley, Reading, Massachusetts, 1979.

- H. R. Lewis "Complexity results for classes of quantification formulas," in *J.CSS 21*, pp. 317–353, 1980.

20.2.12 The theory of reals. We noted in Chapter 9 that it is **NP**-complete to tell whether a set of linear inequalities over the integers is satisfiable, while the same problem for reals is in **P** (recall INTEGER and LINEAR PROGRAMMING, 9.5.34). It is amusing to notice that the complexity of problems relating to the integers and reals exhibit a similar behavior in a much more general setting: While it is undecidable whether a first-order expression over the vocabulary $0, 1, +, \times, <$ is a true property of **N** (recall Coroallary 1 to Theorem 6.3), *it is decidable whether such an expressions is true a property of* $\Re$, *the real numbers.* (To see the significant difference between the two theories, consider

$$\forall x \forall y \exists z [x \geq y \vee (x \leq z \wedge z < y)].)$$

Let THEORY OF REALS be the set of all first-order sentences ϕ over this vocabulary such that $\Re \models \phi$. THEORY OF REALS WITH ADDITION is the subset with no occurrence of multiplication.

To show that THEORY OF REALS WITH ADDITION is decidable, we use a general technique that works in many other cases: *Elimination of quantifiers.* For any expression in prenex form $Q_1 x_1 \ldots Q_n x_n \phi(x_1, \ldots, x_n)$, where ϕ is quantifier-free, we show how to convert it to an equivalent quantifier-free expression. By induction, it suffices to show how we can transform the above expression to an equivalent expression $Q_1 x_1 \ldots Q_{n-1} x_{n-1} \phi'(x_1, \ldots, x_{n-1})$, with ϕ' quantifier-free. In fact, assume Q_n is $\forall$ (otherwise rewrite the expression in terms of $\forall x_n$).

(a) Show that ϕ is a Boolean combination of k atomic expressions of the forms $x_n \triangleright \ell_i(x_1, \ldots, x_{n-1})$, where $\triangleright \in \{=, <, >\}$ and the ℓ_i's are *linear functions with rational coefficients.*

(b) Show that ϕ' above can be taken to be $\bigvee_{t \in T} \phi[x_1 \leftarrow t]$, where T is the set of all n^2 terms $\frac{1}{2}(\ell_i(x_1, \ldots, x_{n-1}) + \ell_j(x_1, \ldots, x_{n-1}))$ for all i, j.

(c) Conclude that THEORY OF REALS WITH ADDITION is in 2-**EXP**. Can you improve this to **EXPSPACE**?

For a weak lower bound, we can show that every problem in **NEXP** reduces to THEORY OF REALS WITH ADDITION. To this end, we construct for each $n \geq 0$ expressions (1) $\mu_n(x, y, z)$, (2) $\xi_n(x, y, z)$, and (3) $\beta_n(x, y)$, with length $\mathcal{O}(n)$, with the indicated free variables. These expressions have the following properties: $\Re$ satisfies them, with real numbers a, b, c replacing variables x, y, z, if and only if, respectively: (1) a is an integer between zero and 2^{2^n}, and $a \cdot b = c$; (2) a and c are integers between zero and 2^{2^n}, and $b^a = c$; and (3) a and b are integers $0 \leq a \leq 2^{(2^n+1)^2} + 1)$, $0 \leq b \leq 2^n$, and the bth bit of a is a one.

(d) Show how to construct these expressions, in logarithmic space in n, by induction on n. (To avoid multiple uses of μ_i etc. in the definition of μ_{i+1} you will have to use the trick in the proof of Theorem 19.1.)

(e) Use these expressions to encode any nondeterministic exponential time computation into a sentence, such that the sentence is in THEORY OF REALS WITH ADDITION if and only if the computation is successful (this is another exponentially dilated version of Cook's theorem).

The complexity of THEORY OF REALS WITH ADDITION can be pinpointed precisely to a complexity class, but not one that we have defined in this book (at least, as far as we know...): Alternating Turing machines with exponential time (so far we would have all of exponential space) *but with only n alternations per computation.* This result is from

- L. Berman "The complexity of logical theories," *Theor. Comp. Science 11,* pp. 71–78, 1980.

Interestingly, THEORY OF REALS (the full language, including multiplication) is also decidable. This is done also by elimination of quantifiers, but now of a much more involved kind—for example, eliminating quantifiers from $\exists x a \cdot x \cdot x + b \cdot x + c = 0$ should produce $b \cdot b \geq 4 \cdot a \cdot c$. This is a classical result due to Alfred Tarski; interestingly, it is still open whether the problem remains decidable if we also allow exponentiation.

If we weaken *number theory* by not allowing exponentiation, we know that the problem is still undecidable (recall the references in Chapter 6). If however we also remove multiplication, we arrive at a fragment of number theory known as *Presburger arithmetic.* This theory is decidable also by elimination of quantifiers, and its complexity is also high, see for example

- M. J. Fischer and M. O. Rabin "Super-exponential complexity of Presburger arithmetic," *Complexity of Computation* (R. M. Karp, ed.), SIAM-AMS Symp. in Applied Mathematics, 1974.

20.2.13 Regular expresion equivalence. The class of *regular expressions* is a language over the alphabet $\{0, 1, \emptyset, \cdot, \cup, ^*\}$, defined as follows: First, the unit-length strings 1, 0, and $\emptyset$ are regular expressions. Next, if ρ and ρ' are regular expressions, then so are $\rho \cdot \rho'$, $\rho \cup \rho'$, and ρ^*. The *semantics* of regular expressions is simple: The meaning of regular expression ρ is a *language* $L(\rho) \subseteq \{0,1\}^*$, defined inductively as follows: First, $L(0) = \{0\}$, $L(1) = \{1\}$, and $L(\emptyset) = \{\}$. Then, $L(\rho \cdot \rho') = L(\rho)L(\rho') = \{xy : x \in L(\rho), y \in L(\rho')\}$, $L(\rho \cup \rho') = L(\rho) \cup L(\rho')$, and $L(\rho^*) = L(\rho)^*$.

(a) Describe $L((0 \cup 1^*)^*)$ and $L((10 \cup 1)^* \cup (11 \cup 0)^*)$. Can you design *finite-state automata*, perhaps nondeterministic (recall Problems 2.8.11 and 2.8.18) that decide these languages?

In fact, a language can be decided by a finite automaton if and only if it is the meaning of a regular expression (this is why we called such languages regular in Problem 2.8.11). One direction is easy:

(b) If ρ is a regular expression, show how to design a nondeterministic finite-state automaton that decides $L(\rho)$. (Obviously, by induction on the structure of ρ.) Can you prove the other direction?

Two regular expressions ρ and ρ' are *equivalent* if $L(\rho) = L(\rho')$. Deciding whether two regular expressions are equivalent is an important computational problem, whose variants are all over the upper complexity spectrum. These variants were explored in a seminal paper

- L. J. Stockmeyer and A. R. Meyer "Word problems requiring exponential time," *Proc. 5th ACM Symp. on the Theory of Computing,* pp. 1–9, 1973.

(c) Show that the problem of deciding whether two regular expressions are equivalent is complete for **PSPACE** *even if one of the expressions is* $\{0,1\}^*$. (To show that it is in **PSPACE** use nondeterminism. To show completeness, express as a regular expression the set of all strings that *do not* encode an accepting in-place computation of machine M on input x.)

(d) Call a regular expression **-free* if it has no occurrences of *, the Kleene star. Show that deciding whether two *-free regular expressions are equivalent is **coNP**-complete. (Membership in **coNP** is not hard. To prove completeness, start with an instance of 3SAT and write a *-free regular expression for the set of truth assignments that *fail* to satisfy it.)

(e) Suppose now that we allow the abbreviation ^{2} (*squaring*) in our regular expressions, where $L(\rho^2) = L(\rho)L(\rho)$. We still do not allow *. Show that the equivalence problem is now **coNEXP**-complete. (With ^{2} we can express nonsatisfying truth assignments of an exponentially long expression, as long as its clauses have a certain regularity. Notice that this is another instance of succinctness increasing the complexity by an exponential.)

(f) Suppose next that both ^{2} and * are allowed. Show that the problem is now complete for *exponential space!* (The succinctness phenomenon again, compare with (c).)

(g) Finally, if we also allow the symbol $\neg$ (semantics: $L(\neg\rho) = \{0,1\}^* - L(\rho)$) then *the equivalence problem is not even elementary.* (Each occurrence of $\neg$ can cause the complexity to be raised by another exponential, intuitively because it may require the conversion of a nondeterministic finite-state automaton to a nondeterministic one, recall Problem 2.8.18 .)

20.2.14 A panorama of complexity classes. There is an amusing and instructive way of looking at all diverse complexity classes discussed in this book from a unified point of view. We have one model of computation: The nondeterministic, polynomial time bounded Turing machine, standardized so that it has precisely two choices at each step (arbitrarily ordered as the first and second choice) and halts after precisely $p(n)$ steps on inputs of length n, where p is bounded by a polynomial. Such a machine N operating on input x produces a computation tree with $2^{p(|x|)}$ leaves, each leaf labeled with a "yes" or "no". Now, since choices are ordered, these leaves are also ordered, and therefore the computation of N on input x can be considered as a string in $\{0,1\}^{2^{p(|x|)}}$, disregarding for a moment the distinction between "yes"-"no" and 1-0. We denote this string as $N(x)$.

A language $L \subseteq \{0,1\}^*$ will be called a *leaf language*. Let A and R be two *disjoint* leaf languages (the *accepting* and *rejecting* leaf language, respectively). Now, any two such languages define a *complexity class:* Let $\mathcal{C}[A,R]$ be the class of all languages L such that there is a (standardized) nondeterministic Turing machine N with the following property: $x \in L$ if and only if $N(x) \in A$, and $x \notin L$ if and only if $N(x) \in R$.

(a) Show that $\mathbf{P} = \mathcal{C}[A,R]$ where $A = 1^*$ and $R = 0^*$. Show that $\mathbf{NP} = \mathcal{C}[A,R]$ where $A = \{0,1\}^*1\{0,1\}^*$ and $R = 0^*$. Show that $\mathbf{RP} = \mathcal{C}[A,R]$ where $A = \{x \in \{0,1\}^* :$ x has more 1's than 0's$\}$ and $R = 0^*$.

(b) Find appropriate leaf languages A and R such that $\mathcal{C}[A, R]$ is: **coNP**, **PP**, **BPP**, **ZPP**, **UP**, $\oplus$**P**, **NP** $\cap$ **coNP**, **NP** $\cup$ **coNP**, **NP** $\cup$ **BPP**.

(c) Repeat for Σ_2**P**, Σ_j**P**, **PSPACE**.

(d) Consider the leaf language A which consists of all strings x with the following property: If x is subdivided into disjoint substrings of length 2^k, where $k = \lceil \log \log x \rceil$, and if these $\frac{|x|}{2^k}$ strings are considered as binary integers, then the largest such integer is odd. Show that the class $\mathcal{C}[A, \overline{A}]$ is Δ_2**P**. (Recall Theorem 17.5 and Problem 17.3.6.)

(e) Show that all leaf languages considered in (a) through (d) above are in **NL**. Show that, if $A, R \in$ **NL**, then $\mathcal{C}[A, R] \subseteq$ **PSPACE**.

(f) Show that if A is an **NL**-complete leaf language, then $\mathcal{C}[A, \overline{A}] =$ **PSPACE**

(g) Find leaf languages A and R such that $\mathcal{C}[A, R] =$ **EXP**. Repeat for **NEXP**.

(h) Which of the pairs of leaf languages A and R considered in (a) through (d) above are complementary, that is, $A \cup R = \{0,1\}^*$? Which can be redefined to be made complementary? (For example, the pair for **P** in (a) is not complementary, but another complementary pair exists.)

Notice the close correlation of the classes whose definitions are via complementary pairs with the classes we have been informally calling *syntactic*, as opposed to *semantic*. In fact, a perfectly reasonable definition of "syntactic class" would be "any class of the form $\mathcal{C}[A, \overline{A}]$."

(j) Define an appropriate class of functions from leaf languages to leaf languages such that the following is true: If f is a function in this class, and A and R are, as usual, disjoint leaf languages, then $f(A), f(R)$ are also disjoint leaf languages, and $\mathcal{C}[A, R] \subseteq \mathcal{C}[f(A), f(R)]$

As it turns out, a formalism closely related to the leaf languages above can be used to systematize proofs of oracle results, see

- D. P. Bovet, P. Crescenzi, and R. Silvestri "A uniform approach to define complexity classes" *Theor. Comp. Sci, 104*, pp. 263–283, 1992.

20.2.15 Networks of queues. Suppose that we are given a *network of queues*, that is, a finite set $V = \{1, \ldots, n\}$ of queues and a set T of *customer types*, where a type $t_i = (P_i, a_i, S_i, w_i)$ is a set $P_i \subseteq V^*$ of *paths* (sequences of queues that are acceptable ways of servicing a customer of this type), an *inter-arrival time distribution* a_i (how often customers of this sort arrive in the system), for each queue $j \in V$ a *service time distribution* S_{ij} (how long such a customer is going to spend in queue j), and a *weight* w_i (how important for the system is this class of customers). The distributions are discrete ones, and are given explicitly in terms of a set of value-probability pair. The problem is to control this system —basically, to decide how to proceed each time a customer arrives or finishes service at a queue—so as to minimize the weighted sum of the expected total waiting times of the customers.

This is a well-known, important, and fantastically hard problem —for example the case of *two* queues ($n = 2$) is already a notoriously difficult problem.

Problem: Formulate the problem precisely, and show that it is **EXP**-complete (use alternating polynomial space).

20.2.16 Interactive proofs and exponential time. Recall the interactive proof systems between Alice and Bob defined in Section 12.2, and shown in Theorem 19.8 to coincide with **PSPACE**. Suppose that we extend this idea to *multiple provers*. That is, the protocol is now between Bob, who as always has polynomial computing powers and randomization, and *several provers* –call them Alice, Amy, Ann, April, and so on—each with exponential powers, and each very interested in convincing Bob that a string x is actually in language L. Bob can now address each of his questions to any one of the provers, and the prover must answer. In fact, for each input x Bob may interact with a number of provers that depends polynomially on $|x|$. Again if $x \in L$ we want Bob to accept x with probability one; if $x \notin L$, then for any possible set of provers we want Bob to accept with probability less than $2^{-|x|}$.

The key feature which makes the situation interesting is that *the provers cannot communicate with each other during the protocol.* If they could, the situation would be identical to the one with a single prover (a gang of conspiring provers behaves exactly like one prover). But the inability of provers to communicate makes it harder for them to fool Bob, and, as we shall see, possibly allows for more interesting and powerful languages to be thus decided.

If a language L can be decided as above, we say that *it has a multiprover interactive proof system;* we write $L \in$ **MIP**. We say that L has an *oracle proof system* if the following holds: There is a randomized oracle Turing machine $M^?$ such that, if $x \in L$ then there is an oracle A such that $M^A(x) =$ "yes" with probability one; and if $x \notin L$, then for all oracles B, $M^B(x) =$ "yes" with probability less than $2^{-|x|}$.

(a) Show that the following are equivalent:
(1) $L \in$ **MIP**.
(2) L has an interactive proof system with *two* provers.
(3) L has an oracle proof system.

(That (2) implies (1) is, of course, trivial. To show that (1) implies (3), think that the provers agree beforehand (as they have to, because of the lack of communication) on all answers each of them will give to any question by Bob; express this protocol as an oracle machine. To show that (3) implies (2), Bob can simulate M^A by asking one of the provers the oracle queries, and at the end asking the second prover a randomly selected query among those asked of the first—to make sure that the first prover is reciting some oracle, and is not basing her answers on the interaction. Repeat enough times. This argument is from

- L. Fortnow, J. Rompel, and M. Sipser "On the power of multiprover interactive protocols," *Proc. 3rd Conference on Structure in Complexity Theory*, pp. 156–161, 1988.)

(b) Based on (a), show that **MIP** $\subseteq$ **NEXP**. (Use (3).)

Surprisingly, *it can be shown that these two classes coincide.* Once more, the power of interactive protocols achieves its limits (compare with Shamir's theorem, Theorem 19.8). This was proved in

- L. Babai, L. Fortnow, and C. Lund "Nondeterministic exponential time has two-prover interactive protocols," *Proc. 31st IEEE Symp. on the Foundations of Computer Science*, pp. 16–25, 1990; also, *Computational Complexity 1*, pp. 3–40, 1991

by extending the "arithmetization" methodology used in the proof of Shamir's theorem to devise an oracle proof system for a version of SUCCINCT 3SAT (Corollary 1 of Theorem 20.2). The proof now is much more sophisticated. An instance of SUCCINCT 3SAT, together with an alleged satisfying truth assignment provided by the oracle, is converted into a long summation, very much as in the proof of Shamir's theorem. If the truth assignment provided by the oracle is a *multilinear* function (a polynomial of degree one in each variable), then a modification of the proof of Shamir's theorem works. Finally, the oracle has to be tested *for multilinearity* —and this turns out to be the heart of the proof.

20.2.17 NEXP and approximability. The next important step in the array of fascinating developments which led to the proof of Theorem 13.13 was the observation that probabilistic interactive proofs are relevant to the approximability of optimization problems; this was first pointed out in

- U. Feige, S. Goldwasser, L. Lovász, S. Safra, and M. Szegedy "Approximating clique is almost NP-complete," *Proc. 32nd IEEE Symp. on the Foundations of Computer Science*, pp. 2–12, 1991.

The idea is simple: Suppose that $L \in \mathbf{NEXP}$; by the previous problem, we can assume that L has an oracle proof system $M^?$. Let $V(x)$ now be the set of all possible accepting computations of $M^?$ on input x —they are exponentially many in $|x|$. Define the following set of edges: $[c, c'] \in E(x)$, where c, c' are computations in $V(x)$, if and only if there is an oracle A that can cause $M^?$ to follow both c and c' (in other words, if c and c' are "compatible"). It turns out that, since $M^?$ is an oracle proof system for L, the maximum clique of the graph $(V(x), E(x))$ is either very small (if $x \notin L$) or very large (in the case $x \in L$).

Problem: Conclude that if the approximation threshold of CLIQUE (or INDEPENDENT SET, for that matter) is strictly less than one, then $\mathbf{EXP} = \mathbf{NEXP}$ (compare with Corollary 2 of Theorem 13.13).

Much of the present chapter has been about ideas and techniques from the $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ problem "scaled up" to exponential time. Going from the above result to

- S. Arora, S. Safra "Probabilistic checking of proofs," *Proc. 33rd IEEE Symp. on the Foundations of Computer Science*, pp. 2-13, 1992,

and ultimately to

- S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy "Proof verification and hardness of approximation problems," *Proc. 33rd IEEE Symp. on the Foundations of Computer Science*, pp. 14–23, 1992

and Theorems 13.12 and 13.3 involved clever arguments for efficiently "scaling down" to the polynomial domain the techniques of arithmetization and multilinear testing; in fact, this scaling-down effort had already started in the paper by Feige et al. referenced above. For a comprehensive account of these techniques see

- M. Sudan *Efficient Checking of Polynomials and Proofs and the Hardness of Approximation Problems*, PhD dissertation, Univ. of California Berkeley, 1992.

Index

ABPP, 474, 475, 480
AC, 385, 386
accepting language, 504
advice string, 277
AL, 400, 401
algorithm, 1, 3, 4, 24
  approximation, 300
  ϵ-approximate, 300
  Las Vegas, 256
  local improvement, 303
  Monte Carlo, 244, 247, 253
  NC, 376
  parallel, 359
  polynomial-time, 6, 11, 13, 137
  pseudopolynomial, 203, 216, 221, 305
  randomized, 244
  RNC, 381
Alice, 279
alphabet, 19
amplifier, 316, 318
ANOTHER HAMILTON CYCLE, 232
AP, 400, 458
APP, 471
approximation algorithm, 300
approximation threshold, 300–302, 304, 305, 309
arithmetical hierarchy, 68
arithmetization, 476, 507
Arthur-Merlin game, 296
ASPACE$(f(n))$, 400
δ-assignment, 263
asymptotic ϵ-approximate algorithm, 323
ATIME$(f(n))$, 400
atomic expression, 87
average-case **NP**-complete problems, 298
average-case analysis of algorithms, 7, 297, 298
axiom, 101, 124
  nonlogical, 104
axiomatic method, 103
axiomatization, 103

BANDWIDTH MINIMIZATION, 215
Bernoulli random variable, 258
BIN PACKING, 204, 323–325
binary representation of integers, 10, 26, 43
binary search, 228, 417
bipartite graph, 11, 213
BISECTION WIDTH, 193, 211
block respecting Turing machine, 157
Blum complexity, 156
board games, 459, 460, 487
Bob, 279

Boolean circuit, 80, 169, 267, 321, 344, 369, 378, 427, 431
  depth, 370
  monotone, 86, 163, 170, 240, 344
  polynomial family, 268, 276, 431
  size, 267, 370
  variable-free, 81
  width, 406
Boolean connectives, 86
Boolean expression, 73
Boolean function, 79
Boolean hierarchy, 434
Boolean logic, 73
Boolean variable, 73
bottleneck, 11
BPP, 259, 263, 269, 272, 429, 430, 433
δ-**BPP**, 263, 264
Brent's principle, 361, 363
budget B, 13

capacity, 8
  of a cut, 16
carry, 364
Chernoff bound, 258
Chinese remainder theorem, 225, 237
Chomsky hierarchy, 66
chordal graph, 213
chromatic number, 214
circuit complexity, 267, 268, 343
circuit see *Boolean circuit*
CIRCUIT SAT, 81, 163, 164, 171
CIRCUIT VALUE, 81, 162, 168, 392, 400
clause, 75
  Horn, 78, 116
CLIQUE, 190, 344, 507
CLIQUE SIZE, 423
closed under reductions, 166, 401, 492
coincidence probability, 260
k-COLORING, 198
3-COLORING, 198, 297
communication complexity, 392
compactness theorem, 85, 111
comparator gate, 390
complement of a language, 142, 219
complete problem, 165, 409
complexity
  average-case, 298
  Blum, 156
  circuit, 267, 268, 343
  communication, 392
  Kolmogorov or descriptive, 52
  space, 35
  time, 29
complexity class, 29, 139
composition of reductions, 164
computation of a Turing machine, 21, 131, 167
computation table, 167
configuration, 21, 28, 38, 45, 128, 147, 398
configuration graph, 147, 398
conjunction, 73
conjunctive normal form, 75
coNP, 219, 235
consistent set, 105, 107
constant restriction of an optimization problem, 324
constant symbol, 86
CONTEXT-FREE EMPTINESS, 391
context-free grammar, 67, 391
context-free language, 67
context-sensitive grammar, 66
context-sensitive language, 66
contradiction, arguing by, 105
Cook's theorem, 171
  weak verifier version, 319
coRP, 256, 272
counting problem, 439
CRITICAL 3-COLORABILITY, 415
CRITICAL HAMILTON PATH, 415
CRITICAL SAT, 415

crossing sequence, 52
CROSSWORD PUZZLE, 211
crude circuit, 344
cryptography, 279
    public-key, 280
    randomized, 286
cubic graph, 233
CYCLE COVER, 211, 440

decision version of an optimization problem, 9, 13
decoding key d, 279
deduction technique, 104
default, 435
    extension, 435
DEFAULT SAT, 435
δ-expander, 317
$\Delta_i\mathbf{P}$, 424, 433, 434
dense language, 336
density, 336
determinant, 241, 242, 367
    symbolic, 242
diagonalization, 60, 145
discrete logarithm, 281, 295
DISJOINT PATHS, 214, 215
disjunction, 73
disjunctive normal form, 75
distributed computation, 481
DOMINATING SET, 208
DP, 412, 413, 433
dual optimization problems, 222, 236
duality, 222, 236

E, 492, 500
edge of a graph, 3
    undirected, 188
Eisenstein's rectangle, 251
elementary language, 498, 499
elimination of quantifiers, 502
empty string ϵ, 21
encoding key e, 279
encrypted message, 279

ϵ-approximate algorithm, 300
    asymptotic, 323
EQUAL OUTPUTS, 234, 240
equality, 86, 97
Erdős-Rado lemma, 345
Euclid's algorithm, 14, 252, 273
EUCLIDEAN STEINER TREE, 209
EUCLIDEAN TSP, 210
EXACT COVER BY 3-SETS, 207
EXACT TSP, 411, 413
existential second-order logic, 113
    Horn, 176
    Krom, 399
EXP, 142, 145, 491, 499, 500
expander, 317
expectation of a random variable, 247
expression
    atomic, 87
    Boolean, 73
    bounded, 126
    first-order, 88
    second-order, 113
EXPSPACE, 497, 499

FACTORING, 230, 237
false, 74
false negative, 244
false positive, 244
family of circuits, 267
    uniform, 269
feasible solution, 236, 299
Fermat test, 247
Fermat witness, 273
Fermat's (little) theorem, 225
finite automaton, 55, 56
first-order logic, 87
FIRST-ORDER SAT, 495
fixpoint logic, 120
flow, 8
FNP, 229, 230, 235
FP, 229, 230, 235
$\mathbf{FP}^{\mathbf{NP}}$, 416, 418

$\mathbf{FP}^{\mathbf{NP}[\log n]}$, 423
FSAT, 228, 230
full gate, 379
function problems, 227
function symbol, 86

Gödel on $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$, 179
gadget, 187, 420
  choice, 194, 443
  choice-consistency, 200
  consistency, 195, 421, 445
  constraint (or clause), 197, 422, 444
gap theorem, 145
gate of a circuit, 80
Gauss' lemma, 249
Gaussian elimination, 242, 272
GEOGRAPHY, 460
GO, 462
goal K, 9
Gödel's completeness theorem, 107
Gödel's incompleteness theorem, 134
grammar, 66
graph, 3, 26, 89, 93
  bipartite, 11, 213
  chordal, 213
  cubic, 233
  interval, 213
  perfect, 214
  periodic, 493
  planar, 210
  representation, 4, 26
  succinct, 493
  symmetric or undirected, 93, 94, 188, 401
GRAPH ISOMORPHISM, 291, 329
GRAPH NONISOMORPHISM, 291
GRAPH RELIABILITY, 441
graph theory, 89, 93, 94, 112, 173, 176
ϕ-GRAPHS, 94, 95, 112, 172
group theory, 103, 136
Grzegorczyk hierarchy, 357

HALTING, 59, 60
HAMILTON CYCLE, 209
HAMILTON PATH, 114, 193
HAMILTON PATH COMPLEMENT, 219
#HAMILTON PATH, 441, 442
HAPPYNET, 231
hash table, 337
Herbrand's theorem, 119
heuristic, 300, 303
hierarchy
  arithmetical, 68
  Boolean, 434
  Chomsky, 66
  exponential, 498, 499
  Grzegorczyk, 357
  nondeterministic space, 155, 500
  nondeterministic time, 155
  polynomial, 433
  space, 145
  time, 145
Hopfield neural net, 232, 239
Horn clause, 78, 116
HORNSAT, 79, 117, 176, 229

Immerman-Szelepscényi theorem, 151
implicant, 75
IN-PLACE ACCEPTANCE, 480
IN-PLACE DIVERGENCE, 481
inconsistent set, 105
INDEPENDENT SET, 188, 208, 210, 213, 307
4-DEGREE INDEPENDENT SET, 190, 313, 318
k-DEGREE INDEPENDENT SET, 190, 309
induction axiom, 126
inherently sequential problem, 365, 381, 389
INTEGER PROGRAMMING, 201, 217, 502

interactive proof, 289
interactive proof system, 289, 475
    multiprover, 507
interval graph, 213
inverse, 367
IP, 290, 475, 480
isolating lemma, 383
isomorphism conjecture, 351

justified generalization, 106

König's lemma, 85
kernel, 208
KNAPSACK, 202, 203, 305
knapsack encryption scheme, 294
Kolmogorov or descriptive complexity, 52
Krom clause, 399
Krom sentence, 399

L, 35, 142, 148, 395, 405, 406
L-reduction, 309
language, 24
    dense, 336
    sparse, 336
    unary, 336, 337
Las Vegas algorithm, 256
law of quadratic reciprocity, 249
leaf language, 504
Legendre symbol $(\cdot|\cdot)$, 249
LINEAR DIVISIBILITY, 236
LINEAR PROGRAMMING, 201, 354, 502
linear speedup, 32
literal, 73
logic, 71
logic programming, 84, 179
logical characterization, 173, 176, 399, 435
$(\log n, 1)$-restricted verifier, 320
LONGEST PATH, 209
Löwenheim-Skolem theorem, 111

MAJSAT, 256
master tour TSP, 435
matching, 11, 12, 17, 301, 381, 440
    weighted, 382
MATCHING, 12, 241, 349, 381, 440
$\oplus$MATCHING, 449
matrix inversion, 367
matrix multiplication, 360
MAX BISECTION, 192, 193
MAX CUT, 191, 303
MAX FLOW, 8, 12, 16, 365, 377, 378, 383
MAX FLOW (D), 9, 391
MAX NAESAT, 318
MAX OUTPUT, 416
MAX2SAT, 186
MAX-WEIGHT SAT, 416
max-flow min-cut theorem, 9, 15, 221, 381
maximal independent set, 389
    lexicographically first, 390
MAXPCP, 327
MAXSAT, 186, 301, 302
MAX SAT SIZE, 423
MAX3SAT, 314
3-OCCURRENCE MAX3SAT, 315, 316, 318
5-OCCURRENCE MAX2SAT, 318
k-MAXGSAT, 302
MAXSNP, 312, 314, 318, 325, 327
MAXSNP-complete problems, 314, 322
MAXSNP$_0$, 312
mental poker, 289
MIN CUT (D), 221
MINIMUM CIRCUIT, 427
MINIMUM COLORING, 324, 327
MINIMUM UNDIRECTED KERNEL, 209, 325
minimum-weight perfect matching, 382
MIP, 506
model, 90, 92, 111, 114
MONOTONE CIRCUIT VALUE, 178

Monte Carlo algorithm, 244, 247, 253
for compositeness, 253, 274
for 2SAT, 247
for REACHABILITY, 404
Monte Carlo oracle algorithm for SAT, 449
multilinear function, 507

NAESAT, 187, 191
NC, 375, 385, 395, 405
$\mathbf{NC}_1$, 386, 395
$\mathbf{NC}_2$, 395
$\mathbf{NC}_i$, 376, 385, 396, 405
NC algorithm, 376
NE, 492
negative example, 346
network, 8
network of queues, 505
NEXP, 491, 499
NL, 142, 148, 395, 398, 405
NODE COVER, 190, 210, 300
4-DEGREE NODE COVER, 318
k-DEGREE NODE COVER, 310, 313
node or vertex of a graph, 3
nondeterministic Turing machine, 45, 171
NP, 46, 142, 148, 166, 171, 173, 181, 235, 257, 272, 319–321, 330, 433, 499
NP-complete problem, 181, 330, 336, 350
$\mathbf{NP} \cap \mathbf{coNP}$, 221, 222, 235, 255, 412
$\mathbf{NSPACE}(f(n))$, 141, 147, 151, 153
$\mathbf{NTIME}(f(n))$, 47, 141, 147, 353
number theory, 88, 91, 111, 123, 132, 503

$\mathcal{O}$-notation, 5
ODD MAX FLOW, 377, 378
ONE-IN-THREE SAT, 207
one-time pad, 280
one-way function, 281, 283, 286
optimization problem, 221, 236, 299, 411
oracle, 339, 340
oracle proof system, 506
oracle results, 343, 344, 351, 352, 436, 499
oracle Turing machine, 339, 417, 506
answer state "yes" or "no", 339
query state, 339
query string, 339
robust, 353

P, 46, 142, 148, 166, 168, 176, 235, 263, 268, 269, 272, 357, 385, 401, 405, 433, 499
padding, 491
padding function, 333
palindrome, 23, 51, 55
parallel algorithm, 359
parallel computation thesis, 389, 398
parallel computer, 359
parallel time, 360
Parseval's theorem, 265, 275
parsimonious reduction, 441
PARTITION, 217
2-PARTITION, 216
3-PARTITION, 216
4-PARTITION, 216
P-complete problems, 168, 377, 378, 391
$\mathbf{PCP}(\log n, 1)$, 320, 327
pebble game, 487
perfect graph, 214
periodic graph, 483
PERIODIC GRAPH COLORING, 483, 486
PERIODIC NOT-ALL-EQUAL SAT, 486
PERIODIC SAT, 485
periodic scheduling, 483
permanent, 440
PERMANENT, 440, 443, 448, 488
PERMANENT MOD 2, 449
PH, 425, 428, 429, 433

$\Pi_i\mathbf{P}$, 424, 427, 433
polyL, 405
planar graph, 210
PLANAR SAT, 210
PLS, 329
plucking, 345
$\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ problem, 13, 45, 137, 148, 319, 329, 377
$\mathbf{P}_{\|}^{\mathbf{NP}}$, 423, 424
$\oplus\mathbf{P}$, 448, 449, 453
Poisson random variable, 469
polynomial circuits, 267, 269, 277, 352, 431
  uniformly, 269
polynomial composition
  left, 154
  right, 154
polynomial hierarchy, 424, 433
  collapse of, 427, 431
polynomial isomorphism, 332
polynomial-time algorithm, 6, 11, 13, 137
polynomial-time approximation scheme, 307, 311, 321, 322
  fully, 307
polynomial-time reduction, 329
positive example, 346
Post's correspondence problem, 67
PP, 256, 257, 448
PRAM (parallel random access machine), 371, 374
  variants, 387, 388
Pratt's theorem, 222
prefix sums, 363
prenex normal form, 99, 100
prime number, 222
prime number theorem, 277
PRIMES, 222, 235, 253, 273, 274
primes, distribution of, 277, 286, 477
primitive root, 224
probabilistic alternating Turing machine, 471
probabilistic method, 270, 317, 430
probabilistically checkable proof, 320, 327
product construction, 307
proper complexity function, 140
protocol, 287
#P, 441–443, 448
pseudopolynomial algorithm, 203, 216, 221, 305
pseudorandom number generators, 276, 295
PSPACE, 142, 148, 150, 340, 429, 433, 456, 458, 460, 471, 475, 499
$\mathbf{PT/WK}(f(n), g(n))$, 370, 373, 375

QSAT, 455, 456, 458, 475
QSAT_i, 427, 428
quantifier, 88, 90, 97, 425, 457, 471, 501
  bounded, 126
  second-order, 114

R, 63, 499
random **NP**-complete problems, 298
random access machine
  configuration, 38
  function computed by a, 39
  input, 38
  instructions, 37
  parallel (PRAM), 371, 374
  program counter, 37
  time spent by a, 39
random access machine (RAM), 36
random oracle hypothesis, 351
random source, 259
  δ-, 261
  slightly, 261
random walk, 245, 272, 402, 407
randomized algorithm, 244
randomized cryptography, 286
Razborov's theorem, 344

RE, 63
REACHABILITY, 3, 40, 49, 112, 114, 120, 162, 362, 398, 500
  undirected, 401, 407
reachability method, 147, 149, 151, 398
REACHABLE DEADLOCK, 482
recursion theorem, 68
recursive function, 24
recursive language, 24, 59–61, 63
recursively enumerable language, 24, 59, 61, 63
recursively inseparable languages, 63
reducible, 160
reduction, 12, 59, 60, 160, 177, 268
  Cook, 177
  Karp, 177
  L-, 309
  logarithmic-space, 160, 177
  nondeterministic or γ-, 235
  parsimonious, 441
  polynomial-time, 177, 329
  randomized, 449
  truth-table, 177
  Turing, 177
register minimization, 157, 488
regular expression equivalence, 503, 504
regular language, 54, 503
rejecting language, 504
relation symbol, 86
residue, 224
resolution, 85, 236
Rice's theorem, 62
Riemann hypothesis, 273
Riemann witness, 274
RL, 402, 405
RNC, 381, 385
RNC
algorithm, 381
root of a polynomial, 226, 243
RP, 254, 256, 267, 272
RSA encryption scheme, 282

SAT, 77, 171
SAT COMPLEMENT, 219
SAT-UNSAT, 413
satisfiable expression, 76, 95
2SAT, 184, 185, 398
3SAT, 183, 189, 191, 193, 199
#SAT, 442
⊕SAT, 448, 449
Savitch's theorem, 149, 457
SC, 405
SCHÖNFINKEL-BERNAYS SAT, 496
search algorithm, 4, 40
  breadth-first, 5, 11
  depth-first, 5, 40
second-order logic, 113, 173, 176
self-reducibility of SAT, 228, 239, 302, 337, 431
semantic class, 255
SET COVERING, 201, 323
SET PACKING, 201
Shamir's theorem, 475
$\Sigma_i\mathbf{P}$, 424–426, 428, 433
$\Sigma - \Pi$ expression, 476
signature, 288
simple expression, 475
sink, 8
SL, 405, 408
slightly random source, 261
$s - m - n$ theorem, 67
SNP, 311
sound and complete proof system, 86, 107
soundness theorem, 107
source, 8
SPACE$(f(n))$, 35, 141, 147, 151, 157
space complexity, 35
space hierarchy theorem, 145
sparse language, 336
speedup
  linear, 32

speedup theorem, 156
SSAT, 470, 472
standard flow, 378
Steiner tree, 209
STEINER TREE, 209, 326
STOCHASTIC SCHEDULING, 470, 472
subgroup, 253
substitutible, 98
substitution, 97
successor relation, 174, 176
SUCCINCT BISECTION WIDTH, 493, 494
succinct certificate or witness, 182
SUCCINCT CIRCUIT SAT, 493, 494
SUCCINCT CIRCUIT VALUE, 493
succinct graph, 493
SUCCINCT HAMILTON PATH, 493, 494
succinct input representation, 493
SUCCINCT NON-EMPTINESS, 500
SUCCINCT REACHABILITY, 500
SUCCINCT 3SAT, 493
sunflower, 345
symmetric Turing machine, 407
SYMMETRY, 94
syntactic class, 255, 505
system of communicating processes, 481

TAXICAB RIPOFF, 209
Tchebychef's theorem, 278
term, 86
TFNP, 230, 235
THEOREMHOOD, 102, 133
THEORY OF REALS WITH ADDITION, 502
THEORY OF REALS, 503
THRESHOLD SAT, 274
TILING, 501
TIME$(f(n))$, 145, 147, 157, 353
time complexity, 29
time hierarchy theorem, 145
Toda's theorem, 448, 452
total functions, 230
tour, 13
tournament, 208
transitive closure, 212
transitive reduction, 212
  strong, 212
trapdoor function, 286
trapdoor predicate, 286
traveling salesman problem (TSP), 13
tree
  rooted, 85
triangle inequality, 305
TRIPARTITE MATCHING, 199
true, 74
truth assignment, 74
  satisfying, 74
TSP, 12, 47, 198, 304, 325, 354, 411, 418
TSP COMPLEMENT, 412
TSP COST, 411
TSP (D), 198
Turing machine, 19
  accepting a language, 24
  accepting state "yes", 19
  alternating, 354, 399
  blank symbol $\sqcup$, 19
  block respecting, 157
  configuration, 21, 28, 45, 128, 147
  cursor, 19, 21
  cursor directions $\leftarrow$, $\rightarrow$, and $-$, 19
  deciding a language, 24, 45, 400
  first symbol $\triangleright$, 19
  halting state h, 19
  nondeterministic, 45, 171
  oblivious, 54
  oracle, 339, 417, 506
  output, 20
  precise, 141, 254, 504
  probabilistic alternating, 471
  rejecting state "no", 19

start state s, 19
state, 19, 167
string, 19, 27
symmetric, 407
transition function, 19, 27
transition relation, 45
two-dimensional, 53
unambiguous, 283
universal, 57, 144
with input and output, 35
with multiple strings, 27, 30
yields relation, 21, 28, 38, 45, 130

UNAMBIGUOUS SAT, 415
unambiguous Turing machine, 283
unary language, 336, 337
unary representation of integers, 10, 26, 384, 423
UNARY TSP, 423
undecidability, 57, 59, 123
UNDIRECTED REACHABILITY, 401, 402
uniform family of circuits, 269
UNIQUE SAT, 415, 433, 434, 453
universal traversal sequence, 407
UP, 382, 383, 453

valid expression, 76, 95
VALIDITY, 102, 133, 219, 220
vocabulary, 87

weak verifier, 319
work by a parallel algorithm, 361

yardstick, 141

zero-knowledge proof, 292
zero-one law, 180
ZIGSAW PUZZLE, 211
ZPP, 256, 272, 273

Author Index

Adleman, L., 235, 273, 294, 296
Aho, A. V., 14, 177, 212, 393
Akl, S., 385
Aleliunas, R., 407
Alexi, W. B., 295
Allender, E., 295
Alon, N., 353
Anderson, R. J., 392
Andreev, A. E., 353
Angluin, D., 452
Appel, K., 180, 214
Arora, S., 328, 507, 508

Babai, L., 296, 452, 507
Baker, T., 351
Balcázar, J. L., 177, 501
Barrington, D., 386
The Beatles, 439, 452
Beaver, D., 488
Bennett, C., 351
Berger, B., 390
Berman, L., 326, 350, 503
Berman, P., 351
Blass, A., 433
Blum, M., 156, 275, 296
Book, R. V., 500
Boole, G., 84
Boppana, R. B., 277, 353
Borodin, A., 155, 405, 408
Bovet, D. P., 505
Brent, R. P., 386
Buss, S. R., 386, 434

Cai, J. -Y., 295, 434, 436, 453
Chandra, A. K., 406
Chistov, A. L., 385
Chomsky, N., 66
Chor, B., 295
Christofides, N., 325
Church, A., 51
Chvátal, V., 323
Cobham, A., 15
Cook, S. A., 55, 155, 178, 388, 389, 391, 406, 408
Cormen, T. H., 14
Crescenzi, P., 505

Dantzig, G. B., 183, 217
Davis, M., 69, 136
Dekhtyar, M. I., 499
Diaz, J., 177
Diffie, W., 294
Dinic, E. A., 16
Dreben, B. S., 501
Dwork, C., 388
Dyer, M. E., 452
Dymond, P. W., 178, 388, 389, 408

Edmonds, J., 15–7, 137, 154
Elias, P., 16
Enderton, H. B., 84, 118
Even, S., 487

Fagin, R., 120, 179
Feige, U., 507
Feigenbaum, J., 488
Feinstein, A., 16
Fenner, S., 352
Fischer, M. J., 155, 276, 503
Ford, L. R., 16
Fortnow, L., 352, 452, 488, 506, 507
Fortune, S., 180, 214
Fraenkel, A. S., 487
Frieze, A. M., 276, 452
Fulkerson, D. R., 16
Furst, M., 387

Gabarró, J., 177
Gabber, O., 327
Galil, Z., 327
Galperin, H., 500
Garey, M. R., 172, 174, 176, 177, 181, 182, 207, 209, 212, 215, 216, 327, 487
Genesareth, M., 436
Gilbert, J. R., 488
Gill, J., 274, 351
Gödel, K., 119, 135
Goldberg, A. V., 17
Goldfarb, W. D., 501
Goldreich, O., 296
Goldschlager, L. M., 178, 391
Goldwasser, S., 295, 296, 507
Golumbic, M. C., 179, 214
Gordreich, O., 295
Gottlob, G., 436
Grädel, E., 179, 408
Graham, R. L., 174, 176, 181, 209, 210, 215
Greenlaw, R., 391
Grollman, J., 295
Grötschel, M., 183, 217, 237
Grzegorczyk, A., 154
Gundermann, T., 434
Gurevich, Y., 298, 433

Haken, A., 236
Haken, W., 180, 214
Halmos, P., 275
Halstenberg, B., 393
Hardy, G. H., 238, 277
Hartmanis, J., 55, 154, 350–352, 434, 500
Håstad, J., 276, 436
Hay, L., 434
Hellman, M. E., 294
Hemachandra, L., 295, 351, 434, 453, 500
Henkin, L., 108, 119
Hennie, F. C., 54
Herbrand, J., 119
Hertz, J., 239, 387
Hilbert, D., 135
Hoover, H. J., 386, 391
Hopcroft, J. E., 14, 16, 51, 157, 180, 214, 488
Huang, M., 273

Immerman, N., 157, 179, 500
Itai, A., 176, 210

Já Já, J., 385
Johnson, D. S., 172–174, 176, 181, 182, 207, 209, 210, 215, 216, 239, 323, 327, 328, 487
Jones, N. D., 391, 406

Kann, V., 326
Kannan, R., 276
Karchmer, M., 394
Karloff, H., 392, 488
Karmarkar, N., 183, 217, 325

Karp, R. M., 16, 172, 178, 207, 277, 325, 385, 407, 437, 487
Khachiyan, L. G., 183, 217
Kirousis, L. M., 392
Klawe, M. M., 386, 394
Kleene, S. C., 51, 68
Knuth, D. E., 14, 15, 181, 215
Ko, K. I., 436
Kolaitis, P., 180
Kolmogorov, A. N., 52
Kou, L., 326
Koutsoupias, E., 297
Kowalski, R. A., 84
Kozen, D. C., 406
Krentel, M. W., 434
Krog, A., 239, 387
Kumar, M. P., 16
Kurtz, S. A., 352

Laaser, W. T., 391, 406
Ladner, R. E., 177, 178, 350
Lagarias, J. C., 276, 294
Lamé, G., 14
Landweber, L., 350
Lautemann, C., 437
Leighton, F. T., 385
Leiserson, C. E., 14
Lengauer, T., 391, 393, 488
Lenstra, A. K., 183, 217, 237, 295
Lenstra, H. W., 183, 217, 237, 295
Levin, L. A., 178, 297
Lewis, H. R., 51, 66, 84, 118, 408, 501
Lewis, P. L. II, 55, 154
Li, M., 52
Lichtenstein, D., 176, 210, 487
Lien, E., 406
Lipton, R. J., 277, 297, 350, 407, 437, 489
Loui, M. C., 389
Lovász, L., 183, 217, 237, 295, 407, 507
Lozano, A., 501
Luby, M., 390
Lund, C., 323, 328, 488, 507, 508
Lynch, J. F., 180, 214
Lynch, N. A., 177

Machtey, M., 135
Mahaney, S. R., 351, 352
Maheshwari, S. N., 16
Malhotra, V. M., 16
Manders, K., 235
Markov, A., 51
Markowsky, G., 326
Matiyasevich, Y., 136
Mayr, E. W., 391, 392
McKenzie, P., 406
Megiddo, N., 174, 208, 239
Merkle, R. C., 294
Meyer, A. R., 155, 487, 503
Micali, S., 295, 296
Miller, G. L., 273
Mostowski, A., 68, 136
Motwani, R., 328, 390, 508
Mulmuley, K., 392

Naor, J., 390
Naor, M., 390
Nielson, N., 436
Nisan, N., 406, 408, 488
Niven, I., 238

Odlyzko, A. M., 294
Orlin, J. B., 489

Palmer, R. G., 239, 387
Papadimitriou, C. H., 14, 51, 174 83, 208, 210, 212, 215, 217, 218, 239, 273, 297, 326, 328, 389, 393, 408, 433–6, 452, 488, 501
Paul, W. J., 157, 353, 394
Pippenger, N., 276, 353, 386, 389, 394

Plumstead, J., 276
Post, E., 51
Pratt, V. R., 238, 388

Rabin, M. O., 15, 154, 273, 503
Rackoff, C., 296, 407
Rajogopalan, S., 298
Ramachandran, V., 385, 389
Razborov, A. A., 353
Reckhow, R. A., 55
Reif, J., 488
Reischuk, R., 388, 393
Reiter, R., 436
Rivest, R. L., 14, 294, 296
Robertson, E., 350
Robertson, N., 180, 215
Robinson, J. A., 86
Robinson, R. M., 136
Rogers, H., 69, 433
Rompel, J., 390, 506
Royer, J. S., 352
Russell, B., 135
Ruzzo, W. L., 388, 391, 407, 408

Safra, S., 328, 507
Santha, M., 275
Savage, J. E., 277
Savitch, W. J., 157, 388
Saxe, J., 387
Schäfer, T. J., 487
Schäffer, A. A., 239
Schnorr, C. P., 239, 295
Schoenfield, J. R., 118
Schöning, U., 350, 352
Schrijver, A., 14, 181, 183, 215, 217, 218, 237
Schwartz, J. T., 273, 488
Seiferas, J., 157
Seiferas, J. I., 155
Selman, A. L., 177, 295
Serna, M. J., 392
Sewelson, V., 434, 500
Seymour, P. D., 180, 215
Shamir, A., 276, 294, 296, 488
Shannon, C., 86
Shanon, C. E., 16
Sharir, M., 488
Shaw, R. A., 391
Shepherdson, J. C., 54, 55
Sideri, M., 436
Silvestri, R., 505
Sipser, M., 56, 179, 277, 296, 351, 353, 387, 393, 437, 487, 506
Skolem, T., 119
Solovay, R., 273, 351
Spira, P. M., 386
Spirakis, P., 392
Staples, J., 391
Stearns, R. E., 54, 55, 154
Steiglitz, K., 14, 183, 218, 237
Stockmeyer, L. J., 120, 176, 210, 387, 388, 406, 434, 487, 503
Strassen, V., 273
Sturgis, H. E., 55
Subramanian, A., 391
Sudan, M., 328, 508
Swart, E. R., 354
Szegedy, M., 328, 507, 508
Szelepcsényi, R., 157
Szemerédi, E., 353, 408
Szwarcfiter, J. L., 176, 210

Tardos, É., 17
Tarjan, R. E., 17, 487, 488
Tarski, A., 136
Toda, S., 452
Tompa, M. L., 408
Torán, J., 501
Trahan, J., 389
Trakhtenbrot, B. A., 15, 155
Trotter, W. T., 353
Turing, A. M., 51

Ullman, J. D., 14, 51, 177, 212, 393

Valiant, L. G., 157, 295, 389, 452
van Emde Boas, P., 51, 389
Vardi, M. Y., 120, 179, 180
Vazirani, U. V., 275, 392
Vazirani, V. V., 275, 392, 452
Venkatesan, R., 298
Vishkin, U., 174, 208, 387
Vitányi, P. M. B., 52
von Neumann, J., 15, 275

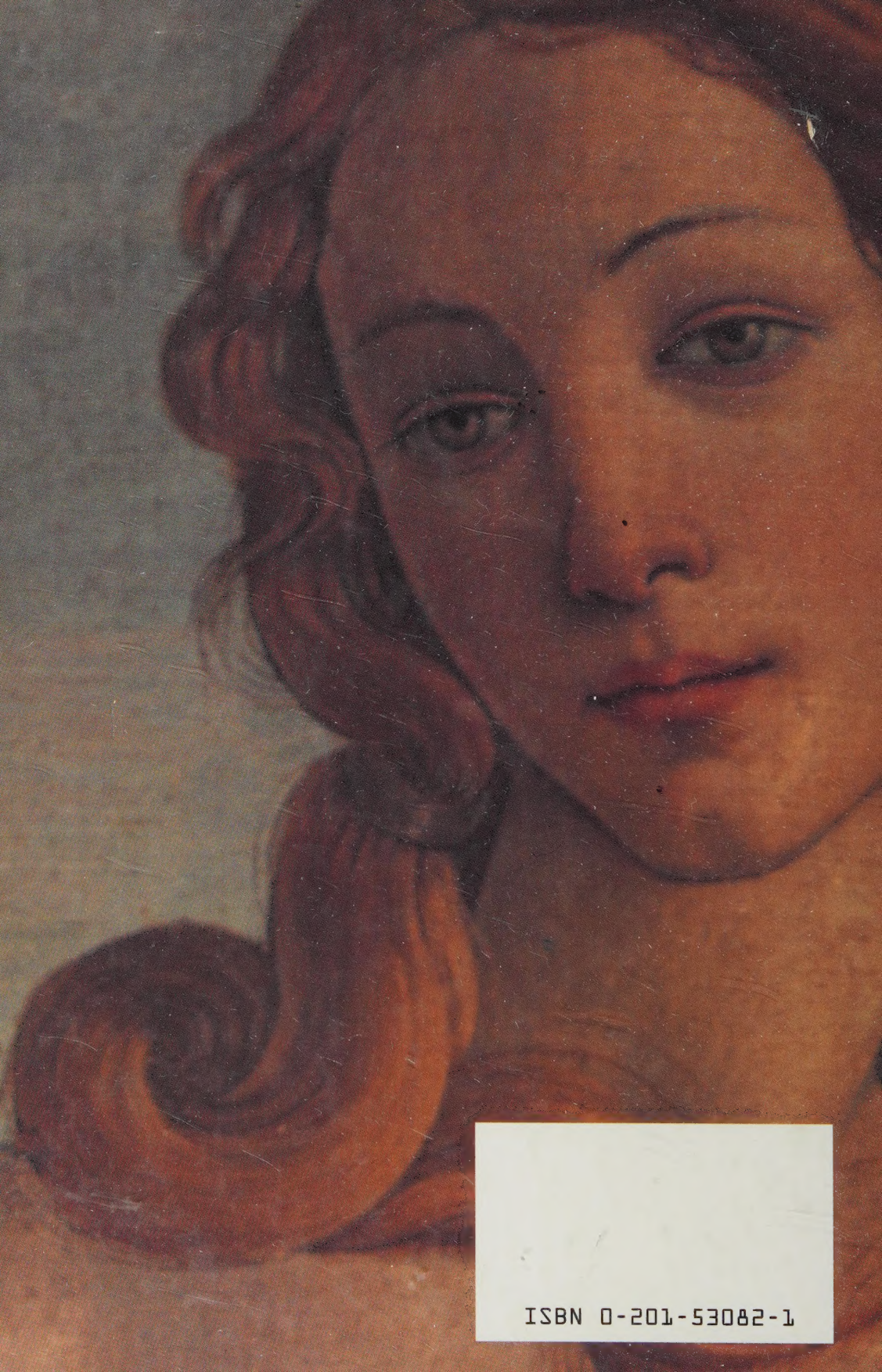
Wagner, K. W., 391, 434
Wechsung, G., 434
Wegener, I., 277
Whitehead, A. N., 135
Wigderson, A., 296, 394, 408, 500
Wittgenstein, L., 73, 84
Wolfe, D., 433
Wrathall, C., 435
Wright, E. M., 238, 277
Wyllie, J., 180, 214

Yannakakis, M., 174, 178, 208, 212, 239, 323, 326, 328, 354, 389, 393, 394, 433, 501
Yao, A. C., 296, 393, 436
Yesha, Y., 487
Young, P. R., 135
Yuval, G., 297

Zachos, S., 275, 452
Zankó, V., 452
Zelikowski, A. Z., 326
Zippel, R. E., 273
Zuckerman, D., 275
Zuckerman, H. S., 238

Lin Tuhen

Sept. 15th. 95



ISBN 0-201-53082-1