Reducibility and Completeness

Chapter 28 of the forthcoming *CRC Handbook on Algorithms and Theory of Computation*

Eric Allender[1]

Rutgers University

Michael C. Loui[2]

University of Illinois at Urbana-Champaign

Kenneth W. Regan[3]

State University of New York at Buffalo

1 Introduction

There is little doubt that the notion of *reducibility* is the most useful tool that complexity theory has delivered to the rest of the computer science community.

For most computational problems that arise in real-world applications, such as the Traveling Salesperson Problem, we still know little about their deterministic time or space complexity. We cannot now tell whether classes such as P and NP are distinct. And yet, even without such hard knowledge, it has been useful in practice to take some new problem A whose complexity needs to be analyzed, and announce that A has roughly the same complexity as Traveling Salesperson, by exhibiting efficient ways of reducing each problem to the other. Thus we can say a lot about problems being equivalent in complexity to each other, even if we cannot pinpoint what that complexity is.

One reason this has succeeded is that, when one partitions the many thousands of real-world computational problems into equivalence classes according to the reducibility relation, there is a surprisingly small number of classes of this partition. Thus, the complexity of almost any problem arising in practice can be classified by showing that it is equivalent to one of a short list of representative problems. It was not originally expected that this would be the case.

[1]Supported by the National Science Foundation under Grant CCR-9509603. Portions of this work were performed while a visiting scholar at the Institute of Mathematical Sciences, Madras, India.
[2]Supported by the National Science Foundation under Grant CCR-9315696.
[3]Supported by the National Science Foundation under Grant CCR-9409104.

Even more amazingly, these "representative problems" correspond in a very natural way to abstract models of computation—that is, they correspond to complexity classes. These classes were defined in the last chapter using a small set of abstract machine concepts: Turing machines, nondeterminism, alternation, time, space. With this and a few simple functions that define time and space bounds, we are able to characterize the complexity of the overwhelming majority of natural computational problems—most of which bear no topical resemblance to any question about Turing machines. This tool has been much more successful than we had any right to expect it would be.

All this leads us to believe that it is no mere accident that problems easily lend themselves to being placed in one class or another. That is, we are disposed to think that these classes really *are* distinct, that the classification is *real*, and that the mathematics developed to deal with them really does describe some important aspect of nature. Nondeterministic Turing machines, with their ability to magically soar through immense search spaces, *seem* to be much more powerful than our mundane deterministic machines, and this reinforces our belief. However, until P vs. NP and similar long-standing questions of complexity theory are completely resolved, our best method of understanding the complexity of real-world problems is to use the classification provided by reducibility, and to trust in a few plausible conjectures.

In this chapter, we discuss reducibility. We will learn about different types of reducibility, and the related notion of *completeness*. It is especially useful to understand NP-completeness. We define NP-completeness precisely, and give examples of NP-complete problems. We show how to prove that a problem is NP-complete, and give some help for coping with NP-completeness. After that, we describe problems that are complete for other complexity classes, under the most efficient reducibility relations. Finally, we cover two other important topics in complexity theory that are motivated by reducibility: relativized computation and the study of sparse languages.

2 Reducibility Relations

In mathematics, as in everyday life, a typical way to solve a new problem is to reduce it to a previously solved problem. Frequently, an instance of the new problem is expressed completely in terms of an instance of the prior problem, and the solution is then interpreted in the terms of the new problem. This kind of reduction is called **many-one reducibility**, and is defined below.

A different way to solve the new problem is to use a subroutine that solves the prior problem.

For example, we can solve an optimization problem whose solution is feasible and maximizes the value of an objective function g by repeatedly calling a subroutine that solves the corresponding decision problem of whether there exists a feasible solution x whose value $g(x)$ satisfies $g(x) \geq k$. This kind of reduction is called **Turing reducibility**, and is also defined below.

Let A_1 and A_2 be languages. A_1 is many-one reducible to A_2, written $A_1 \leq_m A_2$, if there exists a total recursive function f such that for all x, $x \in A_1$ if and only if $f(x) \in A_2$. The function f is called the **transformation function.** A_1 is Turing reducible to A_2, written $A_1 \leq_T A_2$, if A_1 can be decided by a deterministic oracle Turing machine M using A_2 as its oracle, i.e., $A_1 = L(M^{A_2})$. (Recursive functions are defined in Chapter 26, and oracle machines are defined in Chapter 24. The oracle for A_2 models a hypothetical efficient subroutine for A_2.)

If f or M above consumes too much time or space, the reductions they compute are not helpful. To study complexity classes defined by bounds on time and space resources, it is natural to consider resource-bounded reducibilities. Let A_1 and A_2 be languages.

- A_1 is **Karp reducible** to A_2, written $A_1 \leq_m^p A_2$, if A_1 is many-one reducible to A_2 via a transformation function that is computable deterministically in polynomial time.
- A_1 is **Cook reducible** to A_2, written $A_1 \leq_T^p A_2$, if A_1 is Turing reducible to A_2 via a deterministic oracle Turing machine of polynomial time complexity.

The term "polynomial-time reducibility" usually refers to Karp reducibility. If $A_1 \leq_m^p A_2$ and $A_2 \leq_m^p A_1$, then A_1 and A_2 are **equivalent** under Karp reducibility. Equivalence under Cook reducibility is defined similarly.

Karp and Cook reductions are useful for finding relationships between languages of high complexity, but they are not useful at all for distinguishing between problems in P, because all problems in P are equivalent under Karp (and hence Cook) reductions. (Here and later we ignore the special cases $A_1 = \emptyset$ and $A_1 = \Sigma^*$, and consider them to reduce to any language.) To investigate the many interesting complexity classes inside P, we will want to define more restrictive reducibilities, and this we do beginning in Section 5. For the time being, however, we focus on Cook and Karp reducibility.

The key property of Cook and Karp reductions is that they preserve polynomial-time feasibility. Suppose $A_1 \leq_m^p A_2$ via a transformation f. If M_2 decides A_2, and M_f computes f, then to decide whether an input word x is in A_1, we may use M_f to compute $f(x)$, and then run M_2 on input $f(x)$. If the time complexities of M_2 and M_f are bounded by polynomials t_2 and t_f, respectively, then on inputs x of length $n = |x|$, the time taken by this method of deciding A_1 is at most $t_f(n)+t_2(t_f(n))$, which is also a polynomial in n. In summary, if A_2 is feasible, and there is an efficient reduction from A_1 to A_2, then A_1 is feasible. Although this is a simple observation, this fact is important enough to state as a theorem. First, though, we need the concept of "closure."

A class of languages $\mathcal{C}$ is **closed under a reducibility** $\leq_r$ if for all languages A_1 and A_2, whenever $A_1 \leq_r A_2$ and $A_2 \in \mathcal{C}$, necessarily $A_1 \in \mathcal{C}$.

Theorem 2.1 P *is closed under both Cook and Karp reducibility.*

Note that this is an instance of an idea that motivated our identification of P with the class of "feasible" problems in Section 2.2 of Chapter 27, namely that the composition of two feasible functions should be feasible. Similar considerations give us the following theorem.

Theorem 2.2 *Karp reducibility and Cook reducibility are transitive; i.e.:*

1. *If* $A_1 \leq_m^p A_2$ *and* $A_2 \leq_m^p A_3$, *then* $A_1 \leq_m^p A_3$.
2. *If* $A_1 \leq_T^p A_2$ *and* $A_2 \leq_T^p A_3$, *then* $A_1 \leq_T^p A_3$.

We shall see the importance of closure under a reducibility in conjunction with the concept of completeness, which we define in the next section.

3 Complete Languages and Cook's Theorem

Let $\mathcal{C}$ be a class of languages that represent computational problems. A language A_0 is **$\mathcal{C}$-hard** under a reducibility $\leq_r$ if for all A in $\mathcal{C}$, $A \leq_r A_0$. A language A_0 is **$\mathcal{C}$-complete** under $\leq_r$ if A_0 is $\mathcal{C}$-hard, and $A_0 \in \mathcal{C}$. Informally, if A_0 is $\mathcal{C}$-hard, then A_0 represents a problem that is at least as difficult to solve as any problem in $\mathcal{C}$. If A_0 is $\mathcal{C}$-complete, then in a sense, A_0 is one of the most difficult problems in $\mathcal{C}$.

There is another way to view completeness. Completeness provides us with tight lower bounds on the complexity of problems. If a language A is complete for complexity class $\mathcal{C}$, then we have a lower bound on its complexity. Namely, A is as hard as the most difficult problem in $\mathcal{C}$, assuming that the complexity of the reduction itself is small enough not to matter. The lower bound is tight because A is in $\mathcal{C}$; that is, the upper bound matches the lower bound.

In the case $\mathcal{C} = \mathsf{NP}$, the reducibility $\leq_r$ is usually taken to be Karp reducibility unless otherwise stated. Thus we say:

- A language A_0 is **NP-hard** if A_0 is NP-hard under Karp reducibility.
- A_0 is **NP-complete** if A_0 is NP-complete under Karp reducibility.

However, many sources take the term "NP-hard" to refer to Cook reducibility.

Many important languages are now known to be NP-complete. Before we get to them, let us discuss some implications of the statement "A_0 is NP-complete," and also some things this statement doesn't mean.

The first implication is that *if* there exists a deterministic Turing machine that decides A_0 in polynomial time—that is, if $A_0 \in \mathsf{P}$—then because P is closed under Karp reducibility (Theorem 2.1 in Section 2), it would follow that $\mathsf{NP} \subseteq \mathsf{P}$, hence $\mathsf{P} = \mathsf{NP}$. In essence, the question of whether P is the same as NP comes down to the question of whether any particular NP-complete language is in P. Put another way, *all* of the NP-complete languages stand or fall together: if one is in P, then all are in P; if one is not, then all are not. Another implication, which follows by a similar closure argument applied to co-NP, is that if $A_0 \in \text{co-}\mathsf{NP}$ then $\mathsf{NP} = \text{co-}\mathsf{NP}$. It is also believed unlikely that $\mathsf{NP} = \text{co-}\mathsf{NP}$, as was noted in connection with whether all tautologies have short proofs in Section 2.2 of Chapter 27.

A common misconception is that the above property of NP-complete languages is actually their definition, namely: if $A \in \mathsf{NP}$, and $A \in \mathsf{P}$ implies $\mathsf{P} = \mathsf{NP}$, then A is NP-complete. This "definition" is wrong. A theorem due to Ladner [Ladner, 1975b] shows that $\mathsf{P} \neq \mathsf{NP}$ if and only if there exists a language A' in $\mathsf{NP} - \mathsf{P}$ such that A' is not NP-complete. Thus, if $\mathsf{P} \neq \mathsf{NP}$, then A' is a counterexample to the "definition."

Another common misconception arises from a misunderstanding of the statement "If A_0 is NP-complete, then A_0 is one of the most difficult problems in NP." This statement is true on one level: if there is any problem at all in NP that is not in P, then the NP-complete language A_0 is one such problem. However, note that there are NP-complete problems in $\mathsf{NTIME}[n]$—and these problems are, in some sense, much *simpler* than many problems in $\mathsf{NTIME}[n^{10^{500}}]$. We discuss the difficulty of NP-complete problems in more detail below after seeing several examples.

We now prove **Cook's Theorem**, which established the first important NP-complete problem. Recall the definition of SAT, the language of satisfiable Boolean formulas, from Section 2.2 of Chapter 27. In this and later Karp-reduction proofs, we highlight the *construction* of the transformation f, check that the *complexity* of f is polynomial, and verify the *correctness* of the reduction.

Theorem 3.1 (Cook's Theorem) SAT *is* NP*-complete.*

Proof. Let $A \in \mathsf{NP}$. Without loss of generality we may assume that $A \subseteq \{0,1\}^*$. There is a polynomial q and a polynomial-time computable relation R such that for all x,

$$x \in A \iff (\exists y : |y| = q(|x|))\ R(x,y).$$

By the construction in Theorem 3.1 of Chapter 27, there is a polynomial p such that for all n, we can build in time $p(n)$ a Boolean circuit C_n, using only binary NAND gates, that decides R on inputs of length $n + q(n)$. C_n has n input nodes labeled $x_1, \ldots, x_n$ and $q = q(n)$ more input nodes labeled $y_1, \ldots, y_q$. C_n has at most $p(n)$ *wires*, which we label by $w_1, \ldots, w_m$, where $m \leq p(n)$ and w_m is a special wire leading out of the output gate.

Construction. We first write a Boolean formula ϕ_n in the x, y, *and* w variables to express that every gate in C_n functions correctly and C_n outputs 1. For every NAND gate in C_n with incoming wires u, v, and for each outgoing wire w of the gate, we add to ϕ_n the following conjunction of three *clauses*

$$(u \vee w) \wedge (v \vee w) \wedge (\bar{u} \vee \bar{v} \vee \bar{w}).$$

These clauses are satisfied by those assignments to u, v, w such that $w = \neg(u \wedge v)$. Intuitively, they assert that the given NAND gate functions correctly.

Thus ϕ_n has three clauses for every wire w except those wires leading from the inputs, each of which carries the label of the corresponding input variable. Finally for the output wire, ϕ_n has the singleton clause (w_m). So ϕ_n has at most $3p(n) + 1$ clauses in all.

Now given x, we form the desired formula $f(x) = \phi_x$ by building ϕ_n, where $n = |x|$, and simply appending n singleton clauses that force the corresponding assignment to the $x_1, \ldots, x_n$ variables. (For example, if $x = 1001$, append $x_1 \wedge \bar{x}_2 \wedge \bar{x}_3 \wedge x_4$.)

Complexity. C_n is built up in roughly $O(p(n))$ time. Building ϕ_n from C_n, and appending the singleton clauses for x, takes a similar polynomial amount of time.

Correctness. Formally, we need to show that for all x, $x \in A \iff f(x) \in \text{SAT}$. By construction, for all x, $x \in A$ if and only if there exists an assignment to the y variables and to the w variables that satisfies ϕ_x. Hence the reduction is correct. $\square$

A glance at the proof shows that ϕ_x is always a Boolean formula in conjunctive normal form (CNF) with clauses of one, two, or three literals each. By introducing some new "dummy" variables, we can arrange that each clause has exactly three literals. Thus we have actually shown that the following *restricted form* of the satisfiability problem is NP-complete:

3-Satisfiability (3SAT)
Instance: A Boolean expression ϕ in conjunctive normal form with three literals per clause.
Question: Is ϕ satisfiable?

One concrete implication of Cook's Theorem is that if deciding SAT is easy (i.e., in polynomial time), then factoring integers is likewise easy, because the decision version of factoring belongs to NP. (See Chapter 39.) This is a surprising connection between two ostensibly unrelated problems.

The main impact, however, is that once one language has been proved complete for a class such as NP, others can be proved complete by constructing transformations. If A_0 is NP-complete, then to prove that another language A_1 is NP-complete, it suffices to prove that $A_1 \in \mathsf{NP}$, and to construct a polynomial-time transformation that establishes $A_0 \leq_m^p A_1$. Since A_0 is NP-complete, for every language A in NP, $A \leq_m^p A_0$, hence by transitivity (Theorem 2.2 in Section 2), $A \leq_m^p A_1$.

Hundreds of computational problems in many fields of science and engineering have been proved to be NP-complete, almost always by reduction from a problem that was previously known to be NP-complete. We give some practically-motivated examples of these reductions, and also some advice on how to cope with NP-completeness.

4 NP-Complete Problems and Completeness Proofs

This and the next two sections are directed toward practitioners who have a computational problem, don't know how to solve it, and want to know how hard it is—specifically, is it NP-complete, or

NP-hard? The following step-by-step procedure will help in answering these questions, and may help in identifying cases of the problem that are tractable even if the problem is NP-hard for general cases. In brief, the steps are:

1. State the problem in general mathematical terms, and formalize the statement.
2. Ascertain whether the problem belongs to NP.
3. If so, try to find it in a compendium of known NP-complete problems.
4. If you cannot find it, try to construct a *reduction* from a related problem that is known to be NP-complete or NP-hard.
5. Try to identify special cases of your problem that are (i) hard, (ii) easy, and/or (iii) the ones you need. Your work in steps 1.–4. may help you here.
6. Even if your cases are NP-hard, they may still be amenable to direct attack by sophisticated methods on high-powered hardware.

These steps are interspersed with a traditional "theorem–proof" presentation and several long examples, but the same sequence is maintained. We emphasize that trying to do the formalization and proofs asked for in these steps may give you useful positive information about your problem.

Step 1. Give a *formal statement* of the problem. State it without using terms that are specific to your own particular discipline. Use common terms from mathematics and data objects in computer science, e.g. graphs, trees, lists, vectors, matrices, alphabets, strings, logical formulas, mathematical equations. For example, a problem in evolutionary biology that a phylogenist would state in terms of "species" and "characters" and "cladograms" can be stated in terms of trees and strings, using an alphabet that represents the taxonomic characters. Standard notions of size, depth, and distance in trees can express the objectives of the problem.

If your problem involves computing a function that produces a lot of output, look for associated yes/no decision problems, because decision problems have been easier to characterize and classify. For instance, if you need to compute matrices of a certain kind, see whether the essence of your problem can be captured by yes/no questions about the matrices, perhaps about individual entries of them. Many optimization problems looking for a solution of a certain minimum cost or maximum value can be turned into decision problems by including a target cost/value "k" as an input parameter, and phrasing the question of whether a solution exists of cost less than (or value greater than) the target k. Several problems given in the examples below have this form.

It may also help to simplify, even over-simplify, your problem by removing or ignoring some particular elements of it. Doing so may make it easier to ascertain what general category of decision problem yours is in or closest to. In the process, you may learn useful information about the problem that tells you what the effects of those specific elements are—we say some more about this in Section 4.2 below.

Step 2. When you have an adequate formalization, ask first: does your decision problem belong to NP? This is true if and only if candidate solutions that would bring about a "yes" answer can be tested in polynomial time—see the extended discussion in Chapter 27, Section 2.2. If it does belong to NP, that's good news for now! Even if not, you may proceed to determine whether it is NP-hard. The problem may be complete for a class such as PSPACE that contains NP. Examples of such problems are given later in this chapter.

Step 3. See whether your problem is already listed in a compendium of (NP-)complete problems. The book [Garey and Johnson, 1988] lists several hundred NP-complete problems arranged by category. The following is intended as a small representative sample. The first five (together with 3SAT) receive extended treatment in [Garey and Johnson, 1988], while the last five receive comparable treatment here. (The language corresponding to each problem is the set of instances whose answers are "yes.")

VERTEX COVER

Instance: A graph G and an integer k.

Question: Does G have a set W of k vertices such that every edge in G is incident on a vertex in W?

CLIQUE

Instance: A graph G and an integer k.

Question: Does G have a set K of k vertices such that every two vertices in K are adjacent in G?

Hamiltonian Circuit

Instance: A graph G.

Question: Does G have a circuit that includes every vertex exactly once?

3-Dimensional Matching

Instance: Sets W, X, Y with $|W| = |X| = |Y| = q$ and a subset $S \subseteq W \times X \times Y$.

Question: Is there a subset $S' \subseteq S$ of size q such that no two triples in S' agree in any coordinate?

Partition

Instance: A set S of positive integers.

Question: Is there a subset $S' \subseteq S$ such that the sum of the elements of S' equals the sum of the elements of $S - S'$?

Independent Set

Instance: A graph G and an integer k.

Question: Does G have a set U of k vertices such that no two vertices in U are adjacent in G?

Graph Colorability

Instance: A graph G and an integer k.

Question: Is there an assignment of colors to the vertices of G so that no two adjacent vertices receive the same color, and at most k colors are used overall?

Traveling Salesperson (TSP)

Instance: A set of m "cities" $C_1, \ldots, C_m$, with a distance $d(i, j)$ between every pair of cities C_i and C_j, and an integer D.

Question: Is there a tour of the cities whose total length is at most D, i.e., a permutation $c_1, \ldots, c_m$ of $\{1, \ldots, m\}$, such that $d(c_1, c_2) + \cdots + d(c_{m-1}, c_m) + d(c_m, c_1) \leq D$?

KNAPSACK

Instance: A set $U = \{u_1, \ldots, u_m\}$ of objects, each with an integer size $size(u_i)$ and an integer profit $profit(u_i)$, a target size s_0, and a target profit p_0.

Question: Is there a subset $U' \subseteq U$ whose total cost and total profit satisfy

$$\sum_{u_i \in U'} size(u_i) \leq s_0 \qquad and \qquad \sum_{u_i \in U'} profit(u_i) \geq p_0?$$

The languages of all of these problems are easily seen to belong to NP. For example, to show that TSP is in NP, one can build a nondeterministic Turing machine that simply guesses a tour and checks that the tour's total length is at most D.

Some comments on the last two problems are relevant to steps 1. and 2. above. TRAVELING SALESPERSON provides a single abstract form for many concrete problems about sequencing a series of test examples so as to minimize the variation between successive items. The KNAPSACK problem models the filling of a knapsack with items of various sizes, with the goal of maximizing the total value (profit) of the items. Many scheduling problems for multiprocessor computers can be expressed in the form of KNAPSACK instances, where the "size" of an item represents the length of time a job takes to run, and the size of the knapsack represents an available block of machine time.

If yours is on the list of NP-complete problems, you may skip Step 4, and the compendium may give you further information for Steps 5 and 6. You may still wish to pursue Step 4 if you need more study of particular transformations to and from your problem.

If your problem is not on the list, it may still be close enough to one or more problems on the list to help with the next step.

Step 4. Construct a reduction from an already-known NP*-complete problem.* Broadly speaking, Karp reductions come in three kinds.

- A *restriction* from your problem to a special case that is already known to be NP-complete.
- A *minor adjustment* of an already-known problem.
- A *combinatorial transformation.*

The first two kinds of reduction are usually quite easy to do, and we give several examples before proceeding to the third kind.

Example. PARTITION $\leq_m^p$ KNAPSACK, *by restriction:* Given a PARTITION instance with integers s_i, the corresponding instance of KNAPSACK takes $size(u_i) = profit(u_i) = s_i$ (for all i), and sets the targets s_0 and p_0 both equal to $(\sum_i s_i)/2$. The condition in the definition of the KNAPSACK problem of not exceeding s_0 nor being less than p_0 requires that the sum of the selected items meet the target $(\sum_i s_i)/2$ exactly, which is possible if and only if the original instance of PARTITION is solvable.

In this way, the PARTITION problem can be regarded as a *restriction* or special case of the KNAPSACK problem. Note that the reduction itself goes *from* the more-special problem *to* the more-general problem, even though one thinks of the more-general problem as the one being restricted. The implication is that if the restricted problem is NP-hard, then the more-general problem is NP-hard as well, not vice-versa.

Example. HAMILTONIAN CIRCUIT $\leq_m^p$ TSP *by restriction:* Let a graph G be given as an instance of the HAMILTONIAN CIRCUIT problem, and let G have m vertices $v_1, \ldots, v_m$. These vertices become the "cities" of the TSP instance that we build. Now define a distance function d as follows:

$$d(i,j) = \begin{cases} 1 & \text{if } (v_i, v_j) \text{ is an edge in } G \\ m+1 & \text{otherwise.} \end{cases}$$

Set $D = m$. Clearly, d and D can be computed in polynomial time from G. If G has a Hamiltonian circuit, then the length of the tour that corresponds to this circuit is exactly m. Conversely, if there is a tour whose length is at most m, then each step of the tour must have distance 1, not $m+1$. Then each step corresponds to an edge of G, so the corresponding sequence of vertices forms a Hamiltonian circuit in G. Thus the function f defined by $f(G) = (\{\, d(i,j) : 1 \leq i,j \leq m \,\}, D)$ is a polynomial-time transformation from HAMILTONIAN CIRCUIT to TSP.[4]

Minor Adjustments. Here we consider cases where two problems look different but are really closely connected. Consider CLIQUE, INDEPENDENT SET, and VERTEX COVER. A graph G has a clique of size k if and only if its complementary graph G' has an independent set of size k. It follows

[4]Technically we need f to be a function from Σ^* to Σ^*. However, given a string x we can decide in polynomial time whether x encodes a graph G that can be given as an instance of HAMILTONIAN CIRCUIT. If x does not encode a well-formed instance, then define $f(x)$ to be a fixed instance I_0 of TSP for which the answer is "no." Because this sort of thing can generally always be done, we are free to regard the domain of a reduction function f to be the set of "well-formed instances" of the problem we are reducing from. Henceforth we try to ignore such encoding details.

that the function f defined by $f(G,k) = (G',k)$ is a Karp reduction from Independent Set to Clique. To forge a link to the Vertex Cover problem, note that all vertices *not* in a given vertex cover form an independent set, and vice versa. Thus a graph G on n vertices has a vertex cover of size at most k if and only if G has an independent set of size at least $n-k$. Hence the function $g(G,k) = (G, n-k)$ is a Karp reduction from Independent Set to Vertex Cover. (Note that the same f and g also provide reductions from Clique to Independent Set and from Vertex Cover to Independent Set, respectively. This does not happen for all reductions, and gives a sense in which these three problems are unusually close to each other.)

4.1 NP-Completeness by Combinatorial Transformation

The following examples show how the combinatorial mechanism of one problem (here, 3SAT) can be *transformed* by a reduction into the seemingly much different mechanism of another problem.

Theorem 4.1 Independent Set *is* NP*-complete. Hence also* Clique *and* Vertex Cover *are* NP*-complete.*

Proof. We have remarked already that the languages of these three problems belong to NP, and shown already that Independent Set $\leq_m^p$ Clique and Independent Set $\leq_m^p$ Vertex Cover. It suffices to show that 3SAT $\leq_m^p$ Independent Set.

Construction. Let the Boolean formula ϕ be a given instance of 3SAT with variables $x_1, \ldots, x_n$ and clauses $C_1, \ldots, C_m$. The graph G_ϕ we build consists of a "ladder" on $2n$ vertices labeled $x_1, \bar{x}_1, \ldots, x_n, \bar{x}_n$, with edges $(x_i, \bar{x}_i)$ for $1 \leq i \leq n$ forming the "rungs," and m "clause components." Here the component for each clause C_j has one vertex for each literal x_i or $\bar{x}_i$ in the clause, and all pairs of vertices within each clause component are joined by an edge. Finally, each clause-component node with a label x_i is connected by a "crossing edge" to the node with the opposite label $\bar{x}_i$ in the ith "rung," and similarly each occurrence of $\bar{x}_i$ in a clause is joined to the rung node x_i. This finishes the construction of G_ϕ. See Figure 1.

Also set $k = n + m$. Then the reduction function f is defined for all arguments ϕ by $f(\phi) = (G_\phi, k)$.

Complexity. It is not hard to see that f is computable in polynomial time given (a straightforward encoding of) ϕ.

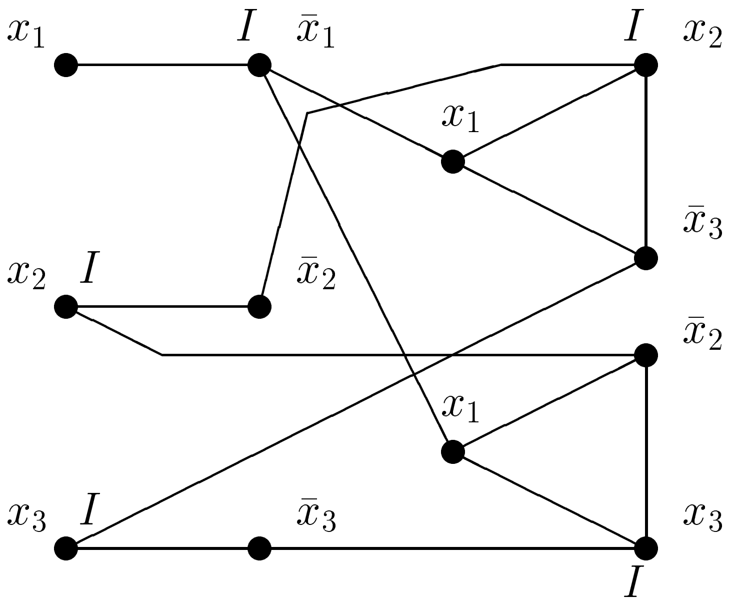

Figure 1: Construction in the proof of NP-completeness of INDEPENDENT SET for the formula $(x_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3)$. The independent set of size 5 corresponding to the satisfying assignment $x_1 =$ `false`, $x_2 =$ `true`, and $x_3 =$ `true` is shown by nodes marked I.

Correctness. To complete the proof, we need to argue that ϕ is satisfiable if and only if G_ϕ has an independent set of size $n + m$. To see this, first note that any independent set I of that size must contain exactly one of the two nodes from each "rung," and exactly one node from each clause component—because the edges in the rungs and the clause component prevent any more nodes from being added. And if I selects a node labeled x_i in a clause component, then I must also select x_i in the ith rung. If I selects $\bar{x}_j$ in a clause component, then I must also select $\bar{x}_j$ in the rung. In this manner I induces a truth assignment in which $x_i = \texttt{true}$ and $x_j = \texttt{false}$, and so on for all variables. This assignment satisfies ϕ, because the node selected from each clause component tells how the corresponding clause is satisfied by the assignment. Going the other way, if ϕ has a satisfying assignment, then that assignment yields an independent set I of size $n + m$ in like manner. □

Since the ϕ in this proof is a 3SAT instance, every clause component is a triangle. The idea, however, also works for CNF formulas with any number of variables in a clause, such as the ϕ_x in the proof of Cook's Theorem.

Now we modify the above idea to give another example of an NP-completeness proof by combinatorial transformation.

Theorem 4.2 GRAPH COLORABILITY *is* NP*-complete*

Proof. *Construction.* Given the 3SAT instance ϕ, we build G_ϕ similarly to the last proof, but with several changes. See Figure 2. On the left, we add a special node labeled "B" and connect it to all $2n$ rung nodes. On the right we add a special node "G" with an edge to B. In any possible 3-coloring of G_ϕ, without loss of generality B will be colored "blue" and the adjacent G will be colored "green." The third color "red" stands for literals made true, whereas green stands for falsity.

Now for each occurrence of a positive literal x_i in a clause, the corresponding clause component has two nodes labeled x_i and x_i' with an edge between them; and similarly an occurrence of a negated literal $\bar{x}_j$ gives nodes $\bar{x}_j$ and $\bar{x}_j'$ with an edge between them. The primed ("inner") nodes in each component are connected by edges into a triangle, but the unprimed ("outer") nodes are not. Each outer node of each clause component is instead connected by an edge to G. Finally, each

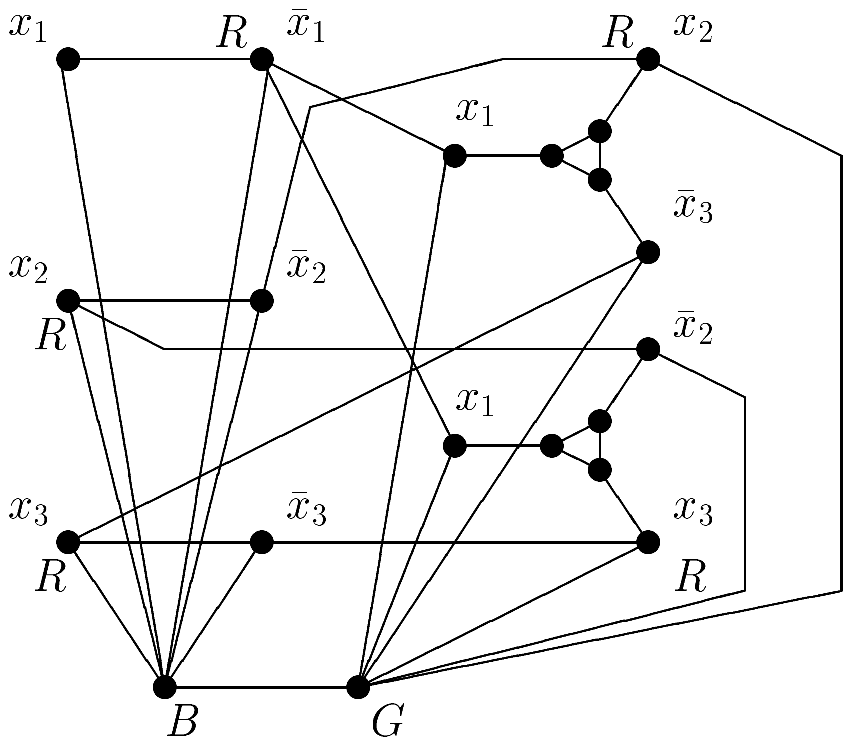

Figure 2: Construction in the proof of NP-completeness of Graph Colorability for the formula $(x_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3)$. The nodes shown colored R correspond to the satisfying assignment $x_1 = \texttt{false}$, $x_2 = \texttt{true}$, and $x_3 = \texttt{true}$, and these together with G and B essentially force a 3-coloring of the graph, which the reader may complete. Note the resemblance to Figure 1.

outer node x_i is connected by a "crossing edge" to the rung node $\bar{x}_i$, and each outer node $\bar{x}_j$ to rung node x_j, exactly as in the INDEPENDENT SET reduction. This finishes the construction of G_ϕ.

Complexity. The function f that given any ϕ outputs G_ϕ, also fixing $k = 3$, is clearly computable in polynomial time.

Correctness. The key idea is that every three-coloring of B, G, and the rung nodes, which corresponds to a truth assignment to the variables of ϕ, can be extended to a 3-coloring of a clause component *if and only if* at least one of the three crossing edges from the component goes to a green rung node. If all three of these edges go to red nodes, then the links to G force each outer node in the component to be colored blue—but then it is impossible to three-color the inner triangle since blue cannot be used. Conversely, any crossing edge to a green node allows the outer node x_i or $\bar{x}_j$ to be colored red, so that one red and two blues can be used for the outer nodes, and this allows the inner triangle to be colored as well. Hence G_ϕ is 3-colorable if and only if ϕ is satisfiable.
□

Note that we have also shown that the restricted form of GRAPH COLORABILITY with k fixed to be 3 (i.e., given a graph G, is G 3-colorable?) is NP-complete. Had we stated the problem this way originally, we would now conclude instead that the more-general graph-colorability problem is NP-complete, similarly to the KNAPSACK and TSP examples above.

Many other reductions from 3SAT use the same basic pattern of a truth-assignment selection component for the variables, components for the clauses (whose behavior depends on whether a variable in the clause is satisfied), and links between these components that make the reduction work correctly. For another example of this pattern, a standard proof that HAMILTONIAN CIRCUIT is NP-complete uses subgraphs V_i for each pair $x_i, \bar{x}_i$ and C_j for each clause. There are two possible ways a circuit can enter V_i, and these correspond to the choices of $x_i = \texttt{true}$ or $x_i = \texttt{false}$ in an assignment. The whole graph is built so that if the circuit enters V_i on the "$x_i = \texttt{true}$" side, then the circuit has the opportunity to visit all nodes in the C_j components for all clauses in which x_i occurs positively, and similarly for occurrences of $\bar{x}_i$ if the circuit enters on the negative side. Hence the circuit can run through every C_j if and only if ϕ is satisfiable. Full details may be found in [Papadimitriou, 1994]. For our last fully-worked-out example, we show a somewhat different pattern in which the individual variables as well as the clauses correspond to top-level components of the following problem.

Disjoint Connecting Paths

Instance: A graph G with two disjoint sets of distinguished vertices $s_1, \ldots, s_k$ and $t_1, \ldots, t_k$, where $k \geq 1$.

Question: Does G contain paths $P_1, \ldots, P_k$, with each P_i going from s_i to t_i, such that no two paths share a vertex?

Theorem 4.3 Disjoint Connecting Paths *is* NP*-complete.*

Proof. First, it is easy to see that Disjoint Connecting Paths belongs to NP: one can design a polynomial-time nondeterministic Turing machine that simply guesses k paths and then deterministically checks that no two of these paths share a vertex. Now let ϕ be a given instance of 3SAT with n variables and m clauses. Take $k = n + m$.

Construction and complexity. The graph G_ϕ we build has distinguished path-origin vertices $s_1, \ldots, s_n$ for the variables and $S_1, \ldots, S_m$ for the clauses of ϕ. G_ϕ also has corresponding sets of path-destination nodes $t_1, \ldots, t_n$ and $T_1, \ldots, T_m$. The other vertices in G_ϕ are nodes u_{ij} for each occurrence of a positive literal x_i in a clause C_j, and nodes v_{ij} for each occurrences of a negated literal $\bar{x}_i$ in C_j. For each i, $1 \leq i \leq n$, G_ϕ is given the edges for a directed path from s_i through all u_{ij} nodes to t_i, and another from s_i through all v_{ij} nodes to t_i. (If there are no occurrences of the positive literal x_i in any clause then the former path is just an edge from s_i right to t_i, and likewise for the latter path if the negated literal $\bar{x}_i$ does not appear in any clause.) Finally, for each j, $1 \leq j \leq m$, G_ϕ has an edge from S_j to every node u_{ij} or v_{ij} for the jth clause, and edges from those nodes to T_j. Clearly these instructions can be carried out to build G_ϕ in polynomial time given ϕ. (See Figure 3.)

Correctness. The first point is that for each i, no path from s_i to t_i can go through both a "u-node" and a "v-node." Setting x_i *true* corresponds to avoiding u-nodes, and setting x_i *false* entails avoiding v-nodes. Thus the choices of such paths for all i represent a truth assignment. The key point is that for each j, one of the three nodes between S_j and T_j will be free for the taking if and only if the corresponding positive or negative literal was made *true* in the assignment, thus satisfying the clause. Hence G_ϕ has the $n + m$ required paths if and only if ϕ is satisfiable. □

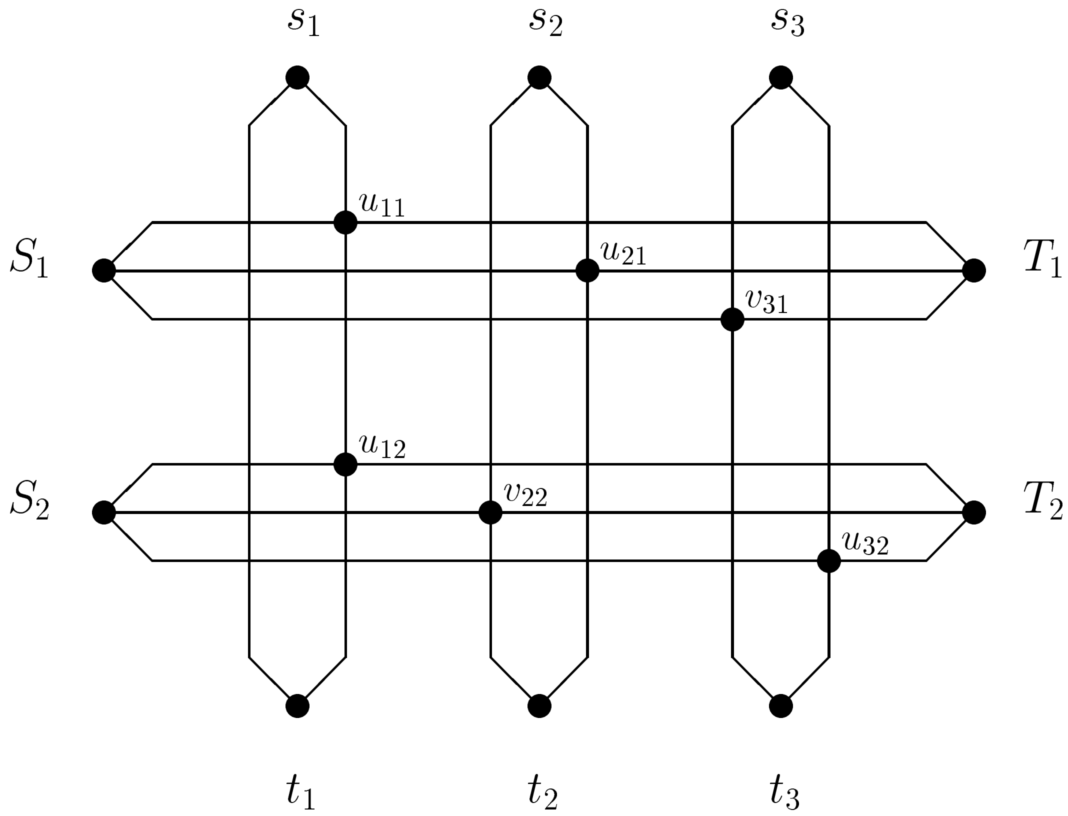

Figure 3: Construction in the proof of NP-completeness of Disjoint Connecting Paths for the formula $(x_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3)$.

4.2 Significance of NP-Completeness

Suppose that you have proved that your problem is NP-complete. What does this mean, and how should you approach the problem now?

Exactly what it means is that your problem does not have a polynomial-time algorithm, *unless* every problem in NP has a polynomial-time algorithm; i.e., unless NP $\neq$ P. We have discussed above the reasons for believing that NP $\neq$ P. In practical terms, you can draw one definite conclusion: Don't bother looking for a "magic bullet" to solve the problem. A simple formula or an easily-tested deciding condition will not be available; otherwise it probably would have been spotted already during the thousands of person-years that have been spent trying to solve similar problems. For example, the NP-completeness of GRAPH 3-COLORABILITY effectively ended hopes that an efficient mathematical formula for deciding the problem would pop out of research on "chromatic polynomials" associated to graphs. Notice that NP-hardness does not say that one needs to be "extra clever" to find a feasible solving algorithm—it says that one probably does not exist at all.

The proof itself means that the combinatorial mechanism of the problem is rich enough to simulate Boolean logic. The proof, however, may also unlock the door to finding saving graces in Steps 5 and 6.

Step 5. Analyze the instances of your problem that are in the range of the reduction. You may tentatively think of these as "hard cases" of the problem. If these differ markedly from the kinds of instances that you expect to see, then this difference may help you refine the statement and conditions of your problem in ways that may actually define a problem in P after all.

To be sure, avoiding the range of one reduction still leaves wide-open the possibility that another reduction will map into your instances of interest. However, it often happens that special cases of NP-complete problems belong to P—and often the boundary between these and the NP-complete cases is sudden and sharp. For one example, consider SAT. The restricted case of three variables per clause is NP-complete, but the case of two variables per clause belongs to P.

For another example, note that the proof of NP-completeness for DISJOINT CONNECTING PATHS given above uses instances in which $k = n+m$; i.e., in which k depends on the number of variables. The case $k = 2$, where you are given G and s_1, s_2, t_1, t_2 and need to decide whether there are vertex-disjoint paths from s_1 to t_1 and from s_2 to t_2, belongs to P. (The polynomial-time algorithm for this case is nontrivial and was not discovered until 1978, as noted in [Garey and Johnson, 1988].)

However, one must also be careful in one's expectations. Suppose we alter the statement of Disjoint Connecting Paths by requiring also that no two vertices in two different paths may have an edge between them. Then the case $k = 2$ of the new problem is NP-complete. (Showing this is a nice exercise; the idea is to make one path climb the "variable ladder" and send the other path through all the clause components.)

4.3 Strong NP-completeness for numerical problems

An important difference between hard and easy cases applies to certain NP-complete problems that involve *numbers*. For example, above we stated that the Partition problem is NP-complete; thus, it is unlikely to be solvable by an efficient algorithm. Clearly, however, we can solve the Partition problem by a simple dynamic programming algorithm, as follows.

For an instance of Partition, let S be a set of positive integers $\{s_1, \ldots, s_m\}$, and let s^* be the total, $s^* = \sum_{i=1}^{m} s_i$. Initialize a linear array B of Boolean values so that $B[0] = \texttt{true}$, and each other entry of B is $\texttt{false}$. For $i = 1$ to m, and for $t = s^*$ down to 0, if $B[t] = \texttt{true}$, then set $B[t + s_i]$ to $\texttt{true}$. After the i^{th} iteration, $B[t]$ is $\texttt{true}$ if and only if a subset of $\{s_1, \ldots, s_i\}$ sums to t. The answer to this instance of Partition is "yes" if $B[s^*/2]$ is ever set to $\texttt{true}$.

The running time of this algorithm depends critically on the representation of S. If each integer in S is represented in binary, then the running time is exponential in the total length of the representation. If each integer is represented in unary—that is, each s_i is represented by s_i consecutive occurrences of the same symbol—then total length of the representation would be greater than s^*, and the running time would be only a polynomial in the length. Put another way, if the *magnitudes* of the numbers involved are bounded by a polynomial in m, then the above algorithm runs in time bounded by a polynomial in m. Since the length of the encoding of such a low-magnitude instance is $O(m \log m)$, the running time is polynomial in the length of the input. The bottom line is that these cases of the Partition problem *are* feasible to solve completely.

A problem is **NP-complete in the strong sense** if there is a fixed polynomial p such that for each instance x of the problem, the value of the largest number encoded in x is at most $p(|x|)$. That is, the integer values are polynomial in the length of the standard representation of the problem. By definition, the 3SAT, Vertex Cover, Clique, Hamiltonian Circuit, and 3–Dimensional Matching problems defined in Section 4 are NP-complete in the strong sense, but Partition and Knapsack are not. The Partition and Knapsack problems can be solved in polynomial time if the integers in their statements are bounded by a polynomial in n—for instance, if numbers are written in unary rather than binary notation.

The concept of strong NP-completeness reminds us that the representation of information can have a major impact on the computational complexity of a problem.

4.4 Coping with NP-hardness

Step 6. Even if you cannot escape NP-hardness, the cases you need to solve may still respond to sophisticated algorithmic methods, possibly needing high-powered hardware.

There are two broad families of direct attack that have been made on hard problems. *Exact solvers* typically take exponential time in the worst case, but provide feasible runs in certain concrete cases. Whenever they halt, they output a correct answer—and some exact solvers also output a proof that their answer is correct. *Heuristic algorithms* typically run in polynomial time in all cases, and often aim to be correct only most of the time, or to find approximate solutions (see Sections 6.1 and 6.2 in the next chapter). They are more common. Popular heuristic methods include genetic algorithms, simulated annealing, neural networks, relaxation to linear programming, and stochastic (Markov) process simulation. Experimental systems dedicated to certain NP-complete problems have recently yielded some interesting results—an extensive survey on solvers for Traveling Salesperson is given by [Johnson and McGeogh, 1997].

There are two ways to attempt to use this research. One is to find a problem close to yours for which people have produced solvers, and try to carry over their methods and heuristics to the specific features of your problem. The other (much more speculative) is to construct a Karp reduction from your problem to their problem, ask to run their program or machine itself on the transformed instance, and then try to map the answer obtained back to a solution of your problem. The hitches are (1) that the currently-known Karp reductions f tend to lose much of the potentially helpful structure of the source instance x when they form $f(x)$, and (2) that approximate solutions for $f(x)$ may map back to terribly sub-optimal or even infeasible answers to x. (See, however, the notion of **L-reductions** in Chapter 29, Section 6.2.) All of this indicates that there is much scope for further research on important practical features of relationships between NP-complete problems. See also Chapter 34 in this *Handbook*.

4.5 Beyond NP-Hardness

If your problem belongs to NP and you cannot prove that it is NP-hard, it may be an "NP-intermediate" problem; i.e., neither in P nor NP-complete. The theorem of Ladner mentioned in Section 3 shows that NP-intermediate problems exist, assuming NP $\neq$ P. However, very few natural problems are currently counted as good candidates for such intermediate status: *factoring*, *discrete logarithm*, *graph-isomorphism*, and several problems relating to *lattice bases* form a very representative list. For the first two, see Chapter 38. The vast majority of natural problems in NP have resolved themselves as being either in P or NP-complete. Unless you uncover a specific connection to one of those four intermediate problems, it is more likely offhand that your problem simply needs more work.

The observed tendency of natural problems in NP to "cluster" as either being in P or NP-complete, with little in between, reinforces the arguments made early in this chapter that P is really different from NP.

Finally, if your problem seems not to be in NP, or alternatively if some more-stringent notion of feasibility than polynomial time is at issue, then you may desire to know whether your problem is *complete for some other complexity class*. We now turn to this question.

5 Complete Problems for NL, P, and PSPACE

We first investigate the log-space analogue of the P vs. NP question, namely whether NL = L. We show that there are natural computational problems that are NL-complete. The question is, under which reducibility? Polynomial-time reducibility is too blunt an instrument here, because NL is contained in P, and so all languages in NL are technically complete for NL under both $\leq_m^p$ and $\leq_T^p$ reductions. We need a reducibility that is fine enough to preserve the distinction between deterministic and nondeterministic log-space that we are attempting to establish and study. The simplest way is to replace the polynomial-time bound in $\leq_m^p$ reductions by a log-space bound.

- A language A_1 is **log-space reducible** to a language A_2, written $A_1 \leq_m^{log} A_2$, if A_1 is many-one reducible to A_2 via a transformation function that is computable by a deterministic Turing machine in $O(\log n)$ space.

There is a log-space analogue of $\leq_T^p$ reducibility, but we do not use it here. Now we show that $\leq_m^{log}$ reductions have the properties we desire:

Theorem 5.1 *(a) (Closure) If $A_1 \leq_m^{log} A_2$ and $A_2 \in \mathsf{L}$, then $A_1 \in \mathsf{L}$.*

(b) (Transitivity) If $A_1 \leq_m^{log} A_2$ and $A_2 \leq_m^{log} A_3$, then $A_1 \leq_m^{log} A_3$.

(c) (Refinement of $\leq_m^p$ reductions) If $A_1 \leq_m^{log} A_2$, then $A_1 \leq_m^p A_2$.

The proof of (a) and (b) is somewhat tricky and rests on the fact that if two functions f and g from strings to strings are computable in log space, then so is the function h defined by $h(x) = g(f(x))$. The hitch is that a log space Turing machine M_f computing f can output the characters of $f(x)$ serially but does not have space to store them. This becomes a problem whenever the machine M_g computing g, whose input is the output from M_f, requests the ith character of $f(x)$, where i may be less than the index j of the previous request. The solution is that since only space and not time is constrained, we may *restart* the computation of $M_f(x)$ from scratch upon the request, and let M_g count the characters that M_f outputs serially until it sees the i^{th} one. Such a counter, and similar ones tracking the movements of M_f's actual input head and M_g's "virtual" input head, can be maintained in $O(\log n)$ space. Thus we need no physical output tape for M_f or input tape for M_g, and we obtain a tandem machine that computes $g(f(x))$ in log space. Part (c) is immediate by the function-class counterpart of the inclusion $\mathsf{L} \subseteq \mathsf{P}$.

The definition of "NL-complete" is an instance of the general definition of completeness at the beginning of Section 3: A language A_1 is NL-complete ("under $\leq_m^{log}$ reductions" is assumed) if $A_1 \in \mathsf{NL}$ and for every language $A_2 \in \mathsf{NL}$, $A_2 \leq_m^{log} A_1$. One defines "P-complete" in a similar manner—again with $\leq_m^{log}$ reductions assumed. Together with the observation that whenever $A_1, A_2 \in \mathsf{L}$ we have $A_1 \leq_m^{log} A_2$ (ignoring technicalities for A_1 or A_2 equal to $\emptyset$ or Σ^*), we obtain a similar state of affairs to what is known about NP-completeness and the P vs. NP question:

Theorem 5.2 *Let A be NL-complete. Then the following statements are equivalent:*

- $\mathsf{NL} = \mathsf{L}$.

- $A \in \mathsf{L}$.

- *Some* NL*-complete language belongs to* L*.*

- *All* NL*-complete languages belong to* L*.*

- *All languages in* L *are* NL*-complete.*

Substitute "P" for "NL" and the same equivalence holds. Note that here we are applying completeness to a class, namely P itself, whose definition does not involve nondeterminism.

Cook's Theorem provides two significant inferences: evidence of intractability, and a connection between computation and Boolean logic. NL-completeness is not to any comparable degree a notion of intractability, but does provide a fundamental link between computations and *graphs*, via the following important problem.

GRAPH ACCESSIBILITY PROBLEM (GAP)
Instance: A directed graph G, and two nodes s and t of G.
Question: Does G have a directed path from node s to node t?

Other names are the *s-t connectivity problem* and the *reachability problem*.

The link involves the concept of an **instantaneous description** (ID) of a Turing machine M. Let us suppose for simplicity that M has just two tapes: one read-only input tape that holds the input x, and one work tape with alphabet $\{0, 1, B\}$, where B is the blank character. Let us also suppose that M never writes a B on its work tape. Then any step of a computation of M on the fixed input x is describable by giving:

- the current state q of M,

- the contents y of the work tape,

- the position i of the input tape head, and

- the position j of the work tape head.

Then the 4-tuple (q, y, i, j) is called an **ID** of M on input x. The restriction on writing B allows us to identify y with a string in $\{0, 1\}^*$. Without loss of generality, we always have $1 \leq i \leq n + 1$, where $n = |x|$, and if $s(n)$ is a space bound on M, also $1 \leq j \leq s(n)$. An ID is also called a **configuration**.

Now define G_x to be the graph whose nodes are all possible IDs of M on input x, and whose directed edges comprise all pairs (I, J) such that M, if set up in configuration I, has a transition that takes it to configuration J in one step. If M is deterministic, then every node in G_x has at most one outgoing arc. Nondeterministic TMs, however, give rise to directed graphs G_x of out-degree more than one. Note that G_x does depend on x, since the step(s) taken from an ID (q, y, i, j) may depend on bit x_i of x.

Theorem 5.3 GAP *is* NL*-complete.*

Proof. GAP belongs to NL because guessing successive edges in a path from 1 to R (when one exists) needs only $O(\log n)$ space to store the label of the current node, and to locate where on the input tape the adjacency information for the current node is stored. To show NL-hardness, let $A \in \mathsf{NL}$. Then A is accepted by a nondeterministic $O(\log n)$ space bounded Turing machine M. We prove that $A \leq_m^{log} GAP$.

Construction: It is easy to modify M to have the properties supposed in the above discussion of IDs and still run in $O(\log n)$ space. We may also code M so that any accepting computation has a final phase that blanks out all used work tape cells, leaves the input head in cell $n + 1$, and halts in a special accepting state q_a. This ensures that every accepting computation (if any) ends in the unique ID $I_t = (q_a, \lambda, n + 1, 1)$.

Now given any x, define G_x as above. Let node s be the unique starting ID $I_s = (q_0, \lambda, 1, 1)$, and let node t be I_t. Note that the size of G_x is polynomial—if M runs in $k \log n$ space, then the size is $O(n^{k+2})$.

Complexity: We show that the transformation f that takes a string x as input and produces the list of edges in G_x as output can be computed by a machine M_f that uses $O(\log n)$ space. For each node $I = (q, y, i, j)$ in turn, M_f reads the i^{th} symbol of x and then produces an edge (I, J) for each J such that M can move from I to J when reading that symbol. The only memory space that M_f needs is the space to step through each I in turn, and count up to the i^{th} input position, and produce each J. $O(\log n)$ space is sufficient for all of this.

Correctness: By the construction, paths in G_x correspond to valid sequences of transitions by M on input x. Hence there exists a path from s to t in G_x if and only if M has an accepting computation on input x. $\square$

To see an example of a reduction between two NL-complete problems, consider the related problem SC of whether a given directed graph is *strongly connected*, meaning that there is a path from every node u to every other node v. Then GAP $\leq_m^{log}$ SC: Take an instance graph G with distinguished nodes s and t and add an edge from every node to s and from t to every node. Computing this transformation needs only $O(\log n)$ space to store the labels of nodes s and t and find their adjacency information on the input tape, changing '0' for "non-edge" to '1' for "edge" as appropriate while writing to the output tape. This transformation is correct because the new edges cannot cause a path from s to t to exist when there wasn't one beforehand, but do allow any such path to be extended to and from any other pair of nodes. SC belongs to NL since a log-space machine can cycle through all pairs (u, v) and nondeterministically guess a path in each case, so SC is NL-complete.

Further variations of the connectivity theme and many other problems are NL-complete. For an interesting contrast to the current situation with NP-completeness, the *complements* of all these problems are also NL-complete under $\leq_m^{log}$ reductions! This is true because NL is closed under complementation (Theorem 2.4 in Chapter 27, Section 2.5). The next problem, however, is apparently harder than the NL-complete problems.

Circuit Value Problem (CVP)

Instance: A Boolean circuit C (see Section 3 of Chapter 27) and an assignment I to the inputs of C.

Question: Does $C(I)$ evaluate to `true`?

Theorem 5.4 CVP *is* P*-complete under* $\leq_m^{log}$ *reductions.*

Proof. That CVP $\in \mathsf{P}$ is clear, and completeness is essentially proved by the construction in Theorem 3.1 of Chapter 27, Section 3.3, which gives a polynomial-size circuit family $\{C_n\}$ that accepts any given language in P. $\square$

Thus CVP belongs to L if and only if $\mathsf{P} = \mathsf{L}$, to NL if and only if $\mathsf{P} = \mathsf{NL}$, and (owing to NC likewise being closed under $\leq_m^{log}$ reductions), to NC if and only if $\mathsf{P} = \mathsf{NC}$. For more P-complete problems, more detail, and discussion of the kind of "intractability" that P-completeness is evidence for, see Chapter 45 in this volume.

The last problem we consider here is a generalization of SAT. *Quantified Boolean formulas* may use the quantifiers $\forall$ and $\exists$ as well as $\{\wedge, \vee, \neg\}$. The formula is *closed* if every variable is quantified. For example, an instance $\phi(x_1, \ldots, x_n)$ of SAT is satisfiable if and only if the closed quantified Boolean formula $\phi' = (\exists x_1)(\exists x_2) \ldots (\exists x_n)\phi$ is true. Another example of a quantified Boolean formula is $\forall x \forall y \exists z (x \wedge (\overline{y} \vee z))$, and this one happens to be false.

QUANTIFIED BOOLEAN FORMULAS (QBF)

Instance: A closed quantified Boolean formula ϕ.

Question: Is ϕ true?

Theorem 5.5 QBF *is* PSPACE*-complete under* $\leq_m^{log}$ *reductions, hence also under* $\leq_m^p$ *reductions.*

Proof. Given an n-variable instance ϕ, $O(n)$ space suffices to maintain a stack with current assignments to each variable, and with this stack one can evaluate ϕ by unwinding one quantifier at a time. So QBF $\in \mathsf{PSPACE}$. For hardness, let $A \in \mathsf{PSPACE}$, and let M be a Turing machine that accepts A in polynomial space. We prove that $A \leq_m^{log}$ QBF.

Construction: Given an input x to M, define the ID graph of G_x as before the proof of Theorem 5.3, but for a polynomial rather than logarithmic space bound. The size of G_x is bounded by $2^{s(n)}$ for some polynomial s, where $n = |x|$. We first define, by induction on r, formulas $\Phi_r(I, J)$ expressing that M started in configuration I can reach configuration J in at most 2^r transitions. The base case formula $\Phi_0(I, J)$ asserts that (I, J) is an edge of G_x.

The idea of the induction is to assert that there is an ID K that is "halfway between" I and J. The straightforward definition $\Phi_r(I, J) := (\exists K)[\Phi_{r-1}(I, K) \wedge \Phi_{r-1}(K, J)]$, however, blows Φ_r up to size exponential in r because of the two occurrences of "Φ_{r-1}" on the right-hand side. The trick is to define, for $r \geq 1$,

$$\Phi_r(I, J) := (\exists K)(\forall I', J') : [(I' = I \wedge J' = K) \vee (I' = K \wedge J' = J)] \rightarrow \Phi_{r-1}(I', J').$$

The single occurrence of "Φ_{r-1}" makes the size of $\Phi_r(I,J)$ roughly proportional to r. Now let I_s be the starting ID of M on input x, and I_t the unique accepting ID, from the proof of Theorem 5.3. Then M accepts x if and only if $\Phi_{s(n)}(I_s, I_t)$ is true.

To convert $\Phi_{s(n)}(I_s, I_t)$ into an equivalent instance ϕ_x of QBF, we can represent IDs by blocks of $s(n)$ Boolean variables. All we need to do is code up polynomially-many instances of the predicates "$I = J$" and "M has a transition from I to J." This is similar to the coding done in the proof of Theorem 3.1, since levels of the circuits C_n in that proof are essentially IDs.

Complexity and Correctness. The only real care needed in the straightforward buildup of $\Phi_{s(n)}(I_1, I_R)$ and then ϕ_x is keeping track of the variables. Since there are only polynomially many of them, each has a tag of length $O(\log n)$, and the housekeeping can be done by a deterministic log-space machine. The reduction is correct since $x \in A$ if and only if $\phi_x \in$ QBF. $\Box$

Remarks. This construction is unaffected if M is nondeterministic, and works for any constructible space bound $s(n) \geq \log n$, producing a formula ϕ_x with $O(s(n)^2)$ Boolean variables. Since ϕ_x can be evaluated deterministically in $O(s(n)^2)$ space, we have also proved Savitch's Theorem (Theorem 2.3(d) in Chapter 27), namely that $\mathsf{NSPACE}[s(n)] \subseteq \mathsf{DSPACE}[s(n)^2]$. We have also essentially proved that PSPACE equals alternating polynomial time (Theorem 2.9b) in Chapter 27), since ϕ_x can be evaluated in $O(s(n)^2)$ *time* by an ATM M that makes existential and universal moves corresponding to the leading "$\exists$" and $\forall$" quantifier blocks in ϕ_x. This also yields an alternative reduction from any language A in PSPACE to QBF, since the proof method for Cook's Theorem (3.1) extends to convert this M directly into a quantified Boolean formula.

One family of PSPACE-complete problems consists of connectivity problems for graphs that, although of exponential size, are specified by a "hierarchical," recursive, or some other scheme that enables one to test whether (u, v) is an edge in time polynomial in the length of the labels of u and v. The graph G_x in the last proof is of this kind, since its edge relation $\Phi_1(I, J)$ is polynomial-time decidable. Another family comprises many two-player combinatorial games, where the question is whether the player to move has a winning strategy. A reduction from QBF to the game question transforms a formula such as $(\exists x)(\forall y)(\exists z)\ldots B$ into reasoning of the form "there exists a move for Black such that for all moves by White, there exists a move for Black such that... Black wins," starting from a carefully constructed position. Decision problems that exhibit this kind of "there exists…for all…" alternation (with polynomially many turns) are often PSPACE-complete.

All PSPACE-complete problems are NP-hard, since $\mathsf{NP} \subseteq \mathsf{PSPACE}$; hence $\mathsf{PSPACE} = \mathsf{P}$ if and only if any one of them belongs to P. The only definite lower bound that follows from the results in this section is that no problem that is PSPACE-complete under $\leq_m^{log}$ reductions belongs to L or even NL, because these classes are closed under $\leq_m^{log}$ reductions and $\mathsf{PSPACE} \neq \mathsf{NL}$. It is, however, still possible to have a problem that is PSPACE-complete under $\leq_m^p$ reductions belong to L, since if $\mathsf{PSPACE} = \mathsf{P}$ then all languages in P are PSPACE-complete under $\leq_m^p$ reductions.

To investigate problems and classes within L, we need even finer reducibility relations than $\leq_m^{log}$. Recent results have brought the reductions defined in the next section to the fore. Amazingly, these new reductions, which are based on our tiniest canonical complexity class, are effective not just within L but for natural problems in all the complexity classes in these chapters, including all the problems defined above.

6 AC^0 Reducibilities

Recall that a function f belongs to AC^0 if and only if the language $\{\langle x, i, b\rangle : i \leq |f(x)| \wedge \text{ bit } i \text{ of } f(x) \text{ equals } b\}$ belongs to AC^0.

- A language A_1 is AC^0 **reducible** to a language A_2, written $A_1 \leq_m^{AC^0} A_2$, if A_1 is many-one reducible to A_2 via a transformation in AC^0.
- A_1 is AC^0**-Turing reducible** to A_2, written $A_1 \leq_T^{AC^0} A_2$, if A_1 is recognized by a DLOGTIME-uniform family of circuits of polynomial size and constant depth, consisting of $\neg$ gates, unbounded fan-in $\wedge$ and $\vee$ gates, and oracle gates for A_2. (An oracle gate for A_2 takes m inputs $x_1, \ldots, x_m$ and outputs 1 if $x_1 \ldots x_m$ is in A_2, and outputs 0 otherwise.)

The next theorem summarizes basic relationships among the five reducibility relations defined thus far.

Theorem 6.1 *For any languages A_1, A_2, and A_3:*

(Transitivity)

(a) If $A_1 \leq_m^{AC^0} A_2$ and $A_2 \leq_m^{AC^0} A_3$, then $A_1 \leq_m^{AC^0} A_3$.

(b) If $A_1 \leq_T^{AC^0} A_2$ *and* $A_2 \leq_T^{AC^0} A_3$*, then* $A_1 \leq_T^{AC^0} A_3$.

(Refinement)

(c) $A_1 \leq_m^{AC^0} A_2 \Longrightarrow A_1 \leq_m^{log} A_2 \Longrightarrow A_1 \leq_m^p A_2 \Longrightarrow A_1 \leq_T^p A_2$.

(d) $A_1 \leq_m^{AC^0} A_2 \Longrightarrow A_1 \leq_T^{AC^0} A_2 \Longrightarrow A_1 \leq_T^p A_2$.

In prose, (c) says that AC^0 reducibility implies log-space reducibility, which implies Karp reducibility, which implies Cook reducibility; and (d) says that AC^0 reducibility implies AC^0-Turing reducibility, which implies Cook reducibility. However, AC^0-Turing reducibility is known *not* to imply log-space reducibility or even Karp reducibility—any language that does not many-one reduce to its complement shows this.

Next, we list which of our canonical complexity classes are closed under which reducibilities. Note that the subclasses of P are not known to be closed under the more powerful reducibilities, and that the nondeterministic time classes are not known to be closed under the Turing reducibilities—mainly because they are not known to be closed under complementation.

Theorem 6.2

(a) P, PSPACE, EXP, *and* $\mathsf{EXPSPACE}$ *are closed under Cook reducibility, and hence under* AC^0 *reducibility,* AC^0*-Turing reducibility, log-space reducibility, and Karp reducibility as well.*

(b) NP *and* NEXP *are closed under Karp reducibility, hence also under* AC^0 *reducibility and log-space reducibility.*

(c) L, NL, *and* NC *are closed under both log-space reducibility and* AC^0*-Turing reducibility, hence also under* AC^0 *reducibility.*

(d) NC^1, TC^0, *and* AC^0 *are closed under* AC^0*-Turing reducibility, hence also under* AC^0 *reducibility.*

For contrast, note that E and NE are *not* closed under AC^0 reducibility—hence they are not closed under any of the other reducibilities, either. To see this, let A be any language in $\mathsf{EXP} - \mathsf{E}$; such languages exist by the time hierarchy theorem (Theorem 2.5 in Chapter 27). Then for some $k > 0$, the language $A_k = \{x10^{|x|^k} \mid x \in A\}$ belongs to E, and it is easy to see that $A \leq_m^{AC^0} A_k$. If

E were closed under $\leq_m^{AC^0}$ reductions, A would be in E, a contradiction. The same can be done for NE using NEXP in place of EXP.

Note that this gives an easy proof that $\mathsf{E} \neq \mathsf{NP}$, because NP has a closure property that E does not share. On the other hand, although this inequality tells us that exactly one of the following must hold:

- $\mathsf{NP} \subset \mathsf{E}$
- $\mathsf{E} \subset \mathsf{NP}$
- $\mathsf{NP} \not\subseteq \mathsf{E}$ and $\mathsf{E} \not\subseteq \mathsf{NP}$,

it is not known *which* of these is true.

6.1 Why Have So Many Kinds of Reducibility?

We have already discussed one reason to consider different kinds of reducibility; in order to explore the important subclasses of P, more restrictive notions such as $\leq_m^{AC^0}$ and $\leq_m^{log}$ are required. However, that does not explain why we study both Karp *and* Cook reducibility, or both AC^0 and AC^0-Turing reductions. It is worth taking a moment to explain this.

If our goal were merely to classify the deterministic complexity of problems, then Cook reducibility ($\leq_T^p$) would be the most natural notion to study. The class of problems that are Cook reducible to A (usually denoted by P^A) characterizes what can be computed quickly if A is easy to compute. Note in particular that A is Cook-reducible to its complement $\overline{A}$, and A and $\overline{A}$ have the same deterministic complexity.

However, if A is an NP-complete language, then A probably does *not* have the same *nondeterministic* complexity as $\overline{A}$. That is, $\overline{A} \in \mathsf{NP}$ only if $\mathsf{NP} = \mathsf{co}\text{-}\mathsf{NP}$. It is worth emphasizing that, if we know only that A is complete for NP under $\leq_T^p$ reductions, the hypothesis $\overline{A} \in \mathsf{NP}$ does not allow us to conclude $\mathsf{NP} = \mathsf{co}\text{-}\mathsf{NP}$. That is, we get *stronger* evidence that $\overline{A}$ has high nondeterministic complexity, if we know that A is complete for NP under the more restrictive kind of reducibility.

This is a general phenomenon: if we know that a language is complete under a more restrictive kind of reducibility, then we know more about its complexity. We have already seen one other example: knowing a language A is complete for PSPACE under $\leq_m^{log}$ reductions tells you that $A \notin \mathsf{NL}$, whereas completeness under $\leq_m^p$ reductions does not even entail that $A \notin \mathsf{L}$. In the latter case we must appeal to the unproven conjecture that $\mathsf{P} \neq \mathsf{PSPACE}$ to infer that A is not in L. For another example, if A is complete for NP under $\leq_m^{AC^0}$ reductions, then we know that $A \notin \mathsf{AC}^0$, whereas if we know only that A is NP-complete under $\leq_m^{log}$ reductions, then we cannot conclude anything about the complexity of A, because we cannot yet rule out the possibility that $\mathsf{L} = \mathsf{NP}$.

6.2 Canonical Classes and Complete Problems

It is an amazing and surprising fact that most computational problems that arise in practice turn out to be complete for some natural complexity class—and complete under some extremely restrictive reducibility such as $\leq_m^{AC^0}$. Indeed, the lion's share of those natural problems known to be complete for NP under $\leq_m^p$ reductions, and for P under $\leq_m^{log}$ reductions, etc., are in fact complete under $\leq_m^{AC^0}$ reductions, too. We observe this after filling out our spectrum of complexity classes with some more decision problems.

INTEGER MULTIPLICATION

Instance: The binary representation of integers x and y, and a number i.

Question: Is the i^{th} bit of the binary representation of $x \cdot y$ equal to 1?

BOOLEAN FORMULA VALUE PROBLEM (BFVP)

Instance: A Boolean formula ϕ and a 0-1 assignment I to the variables in ϕ.

Question: Does $\phi(I)$ evaluate to `true`?

DEGREE-ONE CONNECTIVITY (GAP_1)

Instance: A directed graph in which each node has at most one outgoing edge, and nodes s, t of G.

Question: Is there a path from node s to node t in G?

REGULAR EXPRESSIONS WITH $(\cup, \cdot, ^*)$

Instance: A regular expression α with the standard union, concatenation, and Kleene-star operations (see Chapter 25).

Question: Is there a string that does *not* match α?

REGULAR EXPRESSIONS WITH $(\cup, \cdot, ^2)$

Instance: A regular expression α with union, concatenation, and "squaring" operators (where α^2 denotes $\alpha \cdot \alpha$).

Question: Is there a string that does not match α?

REGULAR EXPRESSIONS WITH $(\cup, \cdot, ^*, ^2)$

Instance: A regular expression α composed of the union, concatenation, "squaring", and Kleene star operators.

Question: Is there a string that does not match α?

N $\times$ N CHECKERS

Instance: A position in checkers played on an $N \times N$ board, with Black to move.

Question: Is this a winning position for Black?

Theorem 6.3 *The following problems are complete for the given complexity classes under $\leq_m^{AC^0}$ reductions, except that* INTEGER MULTIPLICATION *is only known to be complete under $\leq_T^{AC^0}$ reductions.*

- TC^0 : INTEGER MULTIPLICATION.
- NC^1 : BFVP.
- L : GAP_1.
- NL : GAP.
- P : CVP.
- NP : SAT, CLIQUE, VERTEX COVER, *and so on.*
- PSPACE : QBF, REGULAR EXPRESSIONS WITH $(\cup, \cdot, ^*)$.
- EXP : N $\times$ N CHECKERS.
- NEXP : REGULAR EXPRESSIONS WITH $(\cup, \cdot, ^2)$.
- $\mathsf{EXPSPACE}$: REGULAR EXPRESSIONS WITH $(\cup, \cdot, ^*, ^2)$.

The last three problems also belong to E, NE, and $\mathsf{DSPACE}[2^{O(n)}]$ respectively (under suitable encodings), and so they are complete for these respective classes as well. Note that a "tiny" reducibility still gives complete problems for a big class! However, TC^0 is not known to have any complete problems under $\leq_m^{AC^0}$ reductions.

Note that the class NC does not appear anywhere in the list above. NC is not known (or generally believed) to have any complete language under log-space reductions. In fact, if NC does have a language that is complete under $\leq_m^{log}$ reducibility, then there is some k such that $\mathsf{NC}^k = \mathsf{NC}^{k+1} = \ldots = \mathsf{NC}$. This is considered unlikely. This behavior is typical of certain "hierarchy classes," and the *polynomial hierarchy* class PH (defined in the next chapter) behaves similarly with regard to $\leq_m^p$ reductions.

To (im)prove the claim about $\leq_m^{AC^0}$ reductions in Theorem 6.3, we can show that all the reductions in this chapter are computable by uniform AC^0 circuits *without any* $\wedge$ *or* $\vee$ *gates at all!* The circuits have only the constants 0 and 1, the inputs $x_1, \ldots, x_n$, and $\neg$ gates. A function $f : \{0,1\}^* \to \{0,1\}^*$ computed by circuits of this kind is called a *projection*. In a projection, every bit j of the output depends on at most one bit i of the input: it is either always 0, always 1, always x_i, or always the complement of x_i. An AC^0 projection f can be defined without reference to circuits: there is a deterministic Turing machine M that, given binary numbers n and j, decides in $O(\log n)$ time which of these four cases holds, also computing i in the latter two cases. (Note that $m = |f(x)|$ depends only on $n = |x|$; M must also test whether $j > m$ in $O(\log n)$ time.) Now observe:

- Most of the construction of ϕ in our proof of Cook's Theorem depended only on the length n of the argument x. The only dependence on x itself was in the very last piece of ϕ, and under the encoding, x was basically copied bit-by-bit as the signs of the literals. The construction for P-completeness of CVP has the same property.

- In the reductions shown from SAT to graph problems, each *edge* of the target graph depended on only one bit of information of the form, "is variable x_i in clause C_j?" (To meet the technical requirements for projections we must use a convoluted encoding of formulas and graphs, but this is the essential idea.)

- In the construction for NL-completeness of GAP, each edge of G_x depended on what the machine N could do while reading just one bit of x.

- Even in the trickiest proof in this chapter, for PSPACE-completeness of QBF, the only ID whose dependence on x needs to be made explicit is the starting ID I_s, and for this x is just copied bit-by-bit.

Similar ideas work for BFVP, and GAP_1; the reader is invited to investigate the remaining problems.

Our point is not to emphasize projections at the expense of other reductions, but to show that the reductions themselves can be incredibly easy to compute. Thus the complexity levels shown in these completeness results are entirely intrinsic to the target problems. In practice, with classes around P or NP, finding and proving a $\leq_m^p$ or $\leq_m^{log}$ reduction is usually easier, free of encoding fuss, and sufficient for one's purposes.

The list in Theorem 6.3 only begins to illustrate the phenomenon of completeness. Take a computational problem from the practical literature, and chances are it is complete for one of our short list of canonical classes. Here are some more examples, all complete under AC^0 reductions: Does a given deterministic finite automaton M accept a given input x?—L-complete. For certain fixed M, however, the problem is NC^1-complete, and since every regular language belongs to NC^1, this gives a sense in which NC^1 characterizes the complexity of regular languages. Do $N-1$ pairs (i,j) of numbers in $\{1,...,N\}$ form one linked list starting from 1 and ending at N?—L-complete. Various related problems about permutations, list-ranking, depth-first search, and breadth-first search are also L-complete. Satisfiability for 2CNF formulas?—NL-complete. Is $L(G) = \emptyset$ for a given context-free grammar G?—P-complete. Regular Expressions with $(\cup, \cdot)$ only?—NP-complete. Can a multi-threaded finite-state program avert deadlock?—PSPACE-complete. Game problems in which play must halt after polynomially many moves tend to be PSPACE-complete, but if exponentially-long games are possible, as with suitably-generalized versions of Chess and Go as well as Checkers to arbitrarily large boards, they tend to be EXP-complete—and hence intractable to solve!

There are some exceptional problems: connectivity for undirected graphs (L-hard, not known to be in L), matrix determinant, matrix permanent, and the "NP-intermediate" problems mentioned at the end of Section 4.5. (Chapter 29 covers the first one at the end of Section 3, and the next two in Section 7.) But overall, no one would have expected thirty years ago that so many well-studied problems would quantize into so few complexity levels under efficient reductions.

Changing the conditions on a problem also often makes it jump into a new canonical completeness level. The regular-expression problems show this amply. Special cases of NP-complete problems overwhelmingly tend either to remain NP-hard or jump all the way down to P. BFVP is the special case of CVP where every gate in the circuit has fanout 1. Even SAT itself is a restricted case of QBF.

Problems complete for a given class share an underlying mathematical structure that is brought out by the reductions between them. Note that the transformations map tiny local features of one instance x to tiny local features of $f(x)$—particularly when f is a projection! How such local transformations can propagate global decision properties between widely varying problems is a scientific phenomenon that has been studied for itself.

The main significance of completeness, however, is the evidence of intractability it provides. Although in many cases this evidence is based on an unproven conjecture, sometimes it is absolute. Consider the problem Regular Expressions with $(\cup, ^2, \cdot)$, which is complete for NEXP. If this problem were in P, then by closure under Karp reducibility (Theorem 2.1 in Section 2), we would have $\mathsf{NEXP} \subseteq \mathsf{P}$, a contradiction of the Hierarchy Theorems (Theorem 2.5 in Chapter 27). Therefore, this decision problem is infeasible: it has no polynomial-time algorithm. In contrast, decision problems in $\mathsf{NEXP} - \mathsf{P}$ that have been constructed by diagonalization are artificial problems that nobody would want to solve anyway. It is an important point that although diagonalization produces unnatural problems by itself, the combination of diagonalization and completeness shows that *natural* problems are intractable.

However, the next section points out some limitations of current diagonalization techniques.

7 Relativization of the P vs. NP Problem

Let A be a language. Define P^A (respectively, NP^A) to be the class of languages accepted in polynomial time by deterministic (nondeterministic) oracle Turing machines with oracle A.

Proofs that use the diagonalization technique on Turing machines without oracles generally carry over to oracle Turing machines. Thus, for instance, the proof of DTIME hierarchy theorem also shows that, for *any* oracle A, $\mathsf{DTIME}^A[n^2]$ is properly contained in $\mathsf{DTIME}^A[n^3]$. This can be seen as a *strength* of the diagonalization technique, since it allows an argument to "relativize" to computation carried out relative to an oracle. In fact, there are examples of lower bounds (for deterministic, "unrelativized" circuit models) that make crucial use of the fact that the time hierarchies relativize in this sense.

But it can also be seen as a weakness of the diagonalization technique. The following important theorem demonstrates why.

Theorem 7.1 *There exist languages A and B such that $\mathsf{P}^A = \mathsf{NP}^A$, and $\mathsf{P}^B \neq \mathsf{NP}^B$.*

This shows that resolving the P vs. NP question requires techniques that do not relativize, i.e., that do not apply to oracle Turing machines too. Thus, diagonalization as we currently know it is unlikely to succeed in separating P from NP, because the diagonalization arguments we know (and in fact *most* of the arguments we know) relativize. The only major nonrelativizing proof technique in complexity theory appears to be the technique used to prove that $\mathsf{IP} = \mathsf{PSPACE}$. (See Section 5.1 in Chapter 29 and the end notes to that chapter.)

8 Sparse Languages

Despite their variety, the known NP-complete languages are similar in the following sense. Two languages A and B are **P-isomorphic** if there exists a function h such that

- for all x, $x \in A$ if and only if $h(x) \in B$,
- h is bijective (i.e., one-to-one and onto), and
- both h and its inverse h^{-1} are computable in polynomial time.

All known NP-complete languages are P-isomorphic. Thus, in some sense, they are merely different encodings of the same problem. This is yet another example of the "amazing fact" alluded to in Section 6.2, that natural NP-complete languages exhibit unexpected similarities.

Because of this and other considerations, Berman and Hartmanis [Berman and Hartmanis, 1977] conjectured that *all* NP-complete languages are P-isomorphic. This conjecture implies that $\mathsf{P} \neq \mathsf{NP}$, because if $\mathsf{P} = \mathsf{NP}$, then there are finite NP-complete languages, and no infinite language (such as SAT) can be isomorphic to a finite language.

Between the finite languages and the infinite languages lie the sparse languages, which are defined as follows. For a language A over an alphabet Σ, the **census function** of A, denoted $c_A(n)$, is the number of words x in A such that $|x| \le n$. Clearly, $c_A(n) < |\Sigma|^{n+1}$. If $c_A(n)$ is bounded by a polynomial in n, then A is **sparse**. From the definitions, it follows that if A is sparse, and A is P-isomorphic to B, then B is sparse.

If a sparse NP-complete language S exists, then we could use S to solve NP-complete problems efficiently, by the following method. Let A be a language in NP, and let f be a transformation function that reduces A to S in polynomial time $t_f(n)$. To quickly decide membership in A for every word x whose length is at most n, deterministically, compute $f(x)$ and check whether $f(x) \in S$ by looking up $f(x)$ in a table. The table would consist of all words in S whose length is at most $t_f(n)$. The number of entries in this table would be $c_S(t_f(n))$, which is polynomial in n, and hence the total space occupied by the table would be bounded by a polynomial in n.

The Berman-Hartmanis conjecture implies that there is no sparse NP-complete language, however, because SAT is not sparse. A stronger reason for believing there are no such languages is:

Theorem 8.1 *If a sparse* NP*-complete language exists, then* $\mathsf{P} = \mathsf{NP}$.

Very recently, it has been shown that other complexity classes are similarly unlikely to possess sparse complete languages.

Theorem 8.2

1. *If there is a sparse language that is complete for* P *under log-space reducibility, then* $\mathsf{L} = \mathsf{P}$.
2. *If there is a sparse language that is complete for* NL *under log-space reducibility, then* $\mathsf{L} = \mathsf{NL}$.

9 Advice, Circuits, and Sparse Oracles

In the computation of an oracle Turing machine, the oracle language provides assistance in the form of answers to queries, which may depend on the input word. A different kind of assistance, called "advice," depends only on the length of the input word.

Recall the following definitions from Section 3.2 of Chapter 27: A function α is an **advice function** if for every nonnegative integer n, $\alpha(n)$ is a binary word whose length is bounded by a polynomial in n. (An advice function need not be total recursive.)

- $\mathsf{P/poly}$ is the class of languages $A = \{x : \langle x, \alpha(|x|)\rangle \in A'\}$ for some advice function α and some language A' in P,

As was pointed out in that section, $\mathsf{P/poly}$ comprises those problems that can be solved by (non-uniform) circuit families of polynomial size.

Berman and Hartmanis also pointed out the following connection between sparse languages and circuit complexity.

Theorem 9.1 *The following are equivalent:*

1. $A \in \mathsf{P/poly}$.
2. *A is decided by a circuit family of polynomial size complexity.*
3. *$A \in \mathsf{P}^S$ for some sparse language S, that is, A is Cook reducible to a sparse language.*

In Section 8 we discussed the consequences of having a sparse language complete for NP under Karp reducibility. Namely, for any n and for any problem A in NP, there would exist a small table that one could use to efficiently solve A on instances of length at most n. This would also be true if there were a sparse language complete for NP under Cook reducibility. This is equivalent to having $\mathsf{NP} \subseteq \mathsf{P/poly}$. But there is one notable difference between these two situations. By Theorem 8.1, if there is a sparse language complete under Karp reductions, then $\mathsf{P} = \mathsf{NP}$. It is not known if we can conclude that $\mathsf{P} = \mathsf{NP}$ assuming only that there is a sparse language complete under Cook reductions. The reasons for believing that $\mathsf{NP} \not\subseteq \mathsf{P/poly}$ are nearly as strong as those for believing that $\mathsf{NP} \neq \mathsf{P}$; for one, $\mathsf{NP} \subseteq \mathsf{P/poly}$ would imply that there are polynomial-size lookup tables to help one efficiently solve instances of SAT or factor integers, etc. There is other evidence against $\mathsf{NP} \subseteq \mathsf{P/poly}$ that we present in Section 2 in the next chapter.

These connections to circuit complexity and to the isomorphism conjecture have motivated a great deal of research into the complexity of sparse languages under various types of reducibility.

10 Research Issues and Summary

Thanks to the notions of reducibility and completeness, it is possible to give "tight lower bounds" on the complexity of many natural problems, even without yet knowing whether $\mathsf{P} = \mathsf{NP}$. In this chapter, we have seen some examples showing how to prove that problems are NP-complete. We have also explored some of the other notions to which reducibility gives rise, including the notions of relativized computation, P-isomorphism, and the complexity of sparse languages.

There are many natural and important problems that are complete for complexity classes that do not appear in our list of "canonical" complexity classes. In order that these problems can be better understood, it is necessary to introduce some additional complexity classes. That is the topic of the next chapter.

11 Defining Terms

Configuration: For a Turing machine, synonymous with **instantaneous description**.

Cook reduction ($\leq_T^p$): A **reduction** computed by a deterministic polynomial time **oracle Turing machine**.

Cook's theorem: The theorem that the language SAT of satisfiable Boolean formulas (defined in Chapter 27) is **NP-complete**.

Instantaneous description (ID): A string that encodes the current state, head position, and (work) tape contents at one step of a Turing machine computation.

Karp reduction ($\leq_m^p$): A **reduction** given by a polynomial-time computable **transformation function**.

NP: The class of languages accepted by **N**on-deterministic **P**olynomial-time Turing machines. The acronym does not stand for "non-polynomial"—every problem in P belongs to NP.

NP-complete: A language A is NP-complete if A belongs to NP and every language in NP **reduces** to A. Usually this term refers to **Karp reducibility**.

NP-hard: A language A is NP-hard if every language in NP **reduces** to A. Usually this term refers to **Cook reducibility**.

Oracle Turing machine: A Turing machine that may write "query strings" y on a special tape and learn instantly whether y belongs to a language A_2 given as its *oracle*. These are defined in more detail in Chapter 24.

P: The class of languages accepted by (equivalently, decision problems solved by) deterministic polynomial-time Turing machines. Less technically, the class of feasibly solvable problems.

Reduction: A function or algorithm that maps a given instance of a (decision) problem A_1 into one or more instances of another problem A_2, such that an efficient solver for A_2 could be plugged in to yield an efficient solver for A_1.

Sparse language: A language with a polynomially bounded number of strings of any given length.

Transformation function: A function f that maps instances x of one decision problem A_1 to those of another problem A_2 such that for all such x, $x \in A_1 \iff f(x) \in A_2$. (Here we identify a decision problem with the language of inputs for which the answer is "yes.")

References

[Allender, 1990] E. Allender. Oracles versus proof techniques that do not relativize. In *Proc. 1st Annual International Symposium on Algorithms and Computation*, 1990.

[Baker *et al.*, 1975] T. Baker, J. Gill, and R. Solovay. relativizations of the P=NP? question. *SIAM J. Comput.*, 4:431–442, 1975.

[Balcázar *et al.*, 1990] J. Balcázar, J. Díaz, and J. Gabarró. *Structural Complexity I,II.* Springer Verlag, 1990. Part I published in 1988.

[Barrington *et al.*, 1990] D. Mix Barrington, N. Immerman, and H. Straubing. On uniformity within NC^1. *J. Comp. Sys. Sci.*, 41:274–306, 1990.

[Berman and Hartmanis, 1977] L. Berman and J. Hartmanis. On isomorphisms and density of NP and other complete sets. *SIAM J. Comput.*, 6:305–321, 1977.

[Cai and Ogihara, 1997] J.-Y. Cai and M. Ogihara. Sparse hard sets. In L. Hemaspaandra and A. Selman, editors, *Complexity Theory Retrospective II*, pages 53–80. Springer Verlag, 1997.

[Cai and Sivakumar, 1995] J. Cai and D. Sivakumar. The resolution of a Hartmanis conjecture. In *Proc. 36th Annual IEEE Symposium on Foundations of Computer Science*, pages 362–371, 1995. To appear in *J. Comp. Sys. Sci.*.

[Cai *et al.*, 1996] J. Cai, A. Naik, and D. Sivakumar. On the existence of hard sparse sets under weak reductions. In *Proc. 13th Annual Symposium on Theoretical Aspects of Computer Science*, volume 1046 of *Lect. Notes in Comp. Sci.*, pages 307–318. Springer Verlag, 1996.

[Cook, 1971] S. Cook. The complexity of theorem-proving procedures. In *Proc. 3rd Annual ACM Symposium on the Theory of Computing*, pages 151–158, 1971.

[Fortnow, 1994] L. Fortnow. The role of relativization in complexity theory. *Bull. EATCS*, 52:229–244, 1994.

[Garey and Johnson, 1988] M. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, 1988. First edition was 1979.

[Greenlaw *et al.*, 1995] R. Greenlaw, J. Hoover, and W.L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory.* Oxford University Press, 1995. Excerpts available at `http://web.cs.ualberta.ca:80/~hoover/P-complete/`.

[Hartmanis, 1988] J. Hartmanis. New directions in structural complexity theory. In *Proc. 15th Annual International Conference on Automata, Languages, and Programming*, volume 317 of *Lect. Notes in Comp. Sci.*, pages 271–286. Springer Verlag, 1988.

[Hopcroft and Ullman, 1979] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison–Wesley, Reading, MA, 1979.

[Johnson and McGeogh, 1997] D.S. Johnson and L. McGeogh. The traveling salesman problem: a case study in local optimization. In E.H.L. Aarts and J.K. Lenstra, editors, *Local Search in Combinatorial Optimization.* John Wiley and Sons, New York, 1997.

[Jones, 1975] N. Jones. Space-bounded reducibility among combinatorial problems. *J. Comp. Sys. Sci.*, 11:68–85, 1975. Corrigendum *J. Comp. Sys. Sci.* 15:241, 1977.

[Karp, 1972] R. Karp. Reducibility among combinatorial problems. In R.E. Miller and J.W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–104. Plenum Press, 1972.

[Ladner, 1975a] R. Ladner. The circuit value problem is log-space complete for P. *SIGACT News*, 7:18–20, 1975.

[Ladner, 1975b] R. Ladner. On the structure of polynomial-time reducibility. *J. Assn. Comp. Mach.*, 22:155–171, 1975.

[Levin, 1973] L. Levin. Universal sequential search problems. *Problems of Information Transmission*, 9:265–266, 1973.

[Mahaney, 1982] S. Mahaney. Sparse complete sets for NP: solution of a conjecture of Berman and Hartmanis. *J. Comp. Sys. Sci.*, 25:130–143, 1982.

[Papadimitriou, 1994] C. Papadimitriou. *Computational Complexity.* Addison-Wesley, Reading, Mass., 1994.

[Savitch, 1970] W. Savitch. Relationship between nondeterministic and deterministic tape complexities. *J. Comp. Sys. Sci.*, 4:177–192, 1970.

[Schnorr, 1978] C. Schnorr. Satisfiability is quasilinear complete in NQL. *J. Assn. Comp. Mach.*, 25:136–145, 1978.

[Stockmeyer, 1974] L. Stockmeyer. The complexity of decision problems in automata theory and logic. Technical Report MAC-TR-133, Project MAC, M.I.T., Cambridge, Mass., 1974.

[van Melkebeek and Ogihara, 1997] D. van Melkebeek and M. Ogihara. Sparse hard sets for P. In D. Du and K. Ko, editors, *Advances in Complexity and Algorithms.* Kluwer Academic Press, 1997. In press.

[Wagner and Wechsung, 1986] K. Wagner and G. Wechsung. *Computational Complexity.* D. Reidel, 1986.

Further Information.

Cook's Theorem was originally stated and proved for Cook reductions in [Cook, 1971], and later for Karp reductions in [Karp, 1972]. Independently, [Levin, 1973] proved an equivalent theorem using a variant of Karp reductions; sometimes Theorem 3.1 is called the *Cook-Levin theorem*. Our circuit-based proof stems from [Schnorr, 1978], where SAT is shown to be complete for nondeterministic quasi-linear time under deterministic quasi-linear time reductions. The paper [Karp, 1972] showed the large-scale impact of NP-completeness, and Theorems 4.1, 4.2, and 4.3 come from there.

A much more extensive discussion of NP-completeness and techniques for proving problems to be NP-complete may be found in [Garey and Johnson, 1988]. This classic reference also contains a list of hundreds of NP-complete problems. An analogous treatment of problems complete for P can be found in [Greenlaw *et al.*, 1995]—see also Chapter 45 in this volume. Most textbooks on algorithm design or complexity theory also contain a discussion of NP-completeness.

Primary sources for other completeness results in this chapter include: Theorem 5.3 [Savitch, 1970]; Theorem 5.4 [Ladner, 1975a]; Theorem 5.5 [Stockmeyer, 1974]. The paper [Jones, 1975] studied log-space reductions in detail, and also introduced $\leq_m^{AC^0}$ reductions under the name "log-bounded rudimentary reductions." An important paper for the theory of AC^0 is [Barrington *et al.*, 1990], which also discusses the complete problems in for TC^0 and NC^1 in Theorem 6.3. The remaining problems in Theorem 6.3 may be found in [Wagner and Wechsung, 1986]. The texts by Hopcroft and Ullman [Hopcroft and Ullman, 1979] and Papadimitriou [Papadimitriou, 1994] give more examples of problems complete for other classes.

There are many other notions of reducibility, including *truth-table*, *randomized*, and *truth-table* reductions. A good treatment of this material can be found in the two volumes of [Balcázar *et al.*, 1990].

The first relativization results, including Theorem 7.1, were due to [Baker *et al.*, 1975], and many papers have proved more of them. The role of relativization in complexity theory (and even the question of what constitutes a non-relativizing proof technique) is fraught with controversy. Longer discussions of the issues involved may be found in [Allender, 1990, Fortnow, 1994, Hartmanis, 1988].

Sparse languages came to prominence in connection with the Berman-Hartmanis conjecture [Berman and Hartmanis, 1977], where Theorem 9.1 is ascribed to Albert Meyer. Theorem 8.1 is from [Mahaney, 1982], and Theorem 8.2 from [Cai and Sivakumar, 1995, Cai *et al.*, 1996]. Two new surveys of sparse languages and their impact on complexity theory are [Cai and Ogihara, 1997] and [van Melkebeek and Ogihara, 1997].

Information about practical efforts to solve instances of NP-complete and other hard problems is fairly easy to find on the World Wide Web, by searches on problem names such as `Traveling Salesman` (note that variants such as "Salesperson" and the British "Travelling" are also used). Three helpful sites with further links are the Center for Discrete Mathematics and Computer Science (DIMACS), the *TSP Library* (TSPLIB), and the *Genetic Algorithms Archive*;

http://dimacs.rutgers.edu/

http://www.iwr.uni-heidelberg.de/iwr/comopt/soft/TSPLIB95/TSPLIB.html

http://www.aic.nrl.navy.mil:80/galist/

are the current URLs. TSPLIB has down-loadable test instances of the TSP problem drawn mostly from practical sources. There is also an extensive bibliography on the TSP problem called TSPBIB, maintained by P. Moscato at `http://www.ing.unlp.edu.ar/cetad/mos/TSPBIB_home.html`.

See also the **Further Information** section of Chapter 29.