Other Complexity Classes and Measures

Chapter 29 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

In the previous two chapters, we have

- introduced the basic complexity classes,
- summarized the known relationships between these classes, and
- seen how reducibility and completeness can be used to establish tight links between natural computational problems and complexity classes.

Some natural problems seem not to be complete for any of the complexity classes we have seen so far. For example, consider the problem of taking as input a graph G and a number k, and deciding whether k is exactly the length of the shortest traveling salesperson's tour. This is clearly related to the TSP problem discussed in Chapter 28, Section 3, but in contrast to TSP, it seems not to belong to NP, and also seems not to belong to $\mathsf{co\text{-}NP}$.

To classify and understand this and other problems, we will introduce a few more complexity classes. We cannot discuss all of the classes that have been studied—there are further pointers to the literature at the end of this chapter. Our goal is to describe some of the most important classes, such as those defined by probabilistic and interactive computation.

[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.

A common theme is that the new classes arise from the interaction of complexity theory with other fields, such as randomized algorithms, formal logic, combinatorial optimization, and matrix algebra. Complexity theory provides a common formal language for analyzing computational performance in these areas. Other examples can be found in other chapters of this *Handbook*.

2 The Polynomial Hierarchy

Recall from Chapter 27, Section 2.8 that PSPACE is equal to the class of languages that can be recognized in polynomial time on an alternating Turing machine, and that NP corresponds to polynomial time on a nondeterministic Turing machine, which is just an alternating Turing machine that uses only existential states. Thus, in some sense, NP sits near the very "bottom" of PSPACE, and as we allow more use of the power of alternation, we slowly climb up toward PSPACE.

Many natural and important problems reside near the bottom of PSPACE in this sense, but are neither known nor believed to be in NP. (We shall see some examples later in this chapter.) Most of these problems can be accepted quickly by alternating Turing machines that make only two or three alternations between existential and universal states. This observation motivates the definition in the next paragraph.

With reference to Section 2.4 of Chapter 24, define a k**-alternating Turing machine** to be a machine such that on every computation path, the number of changes from an existential state to universal state, or from a universal state to an existential state, is at most $k-1$. Thus, a nondeterministic Turing machine, which stays in existential states, is a 1-alternating Turing machine.

It turns out that the class of languages recognized in polynomial time by 2-alternating Turing machines is precisely $\mathsf{NP}^{\mathrm{SAT}}$. This is a manifestation of something more general, and it leads us to the following definitions.

Let $\mathcal{C}$ be a class of languages. Define

- $\mathsf{NP}^{\mathcal{C}} = \bigcup_{A\in\mathcal{C}} \mathsf{NP}^{A}$,
- $\sum_0^P = \prod_0^P = \mathsf{P}$;

and for $k \geq 0$, define

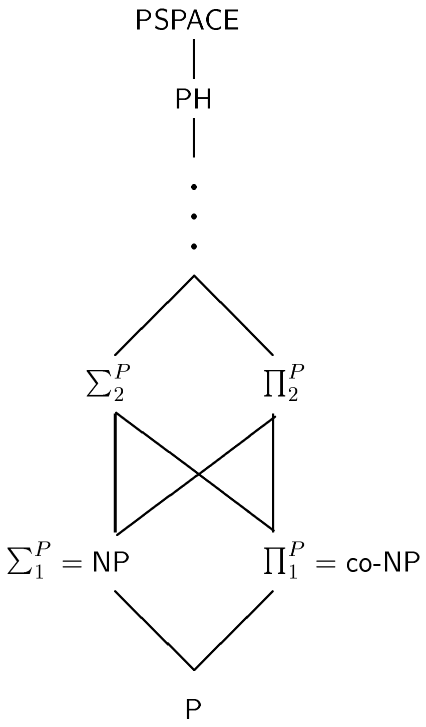

Figure 1: The polynomial hierarchy.

- $\Sigma_{k+1}^P = \mathsf{NP}^{\Sigma_k^P}$,
- $\Pi_{k+1}^P = \text{co-}\Sigma_{k+1}^P$.

Observe that $\Sigma_1^P = \mathsf{NP}^{\mathsf{P}} = \mathsf{NP}$, because each of polynomially many queries to an oracle language in P can be answered directly by a (nondeterministic) Turing machine in polynomial time. Consequently, $\Pi_1^P = \text{co-}\mathsf{NP}$. For each k, $\Sigma_k^P \subseteq \Sigma_{k+1}^P$, and $\Pi_k^P \subseteq \Sigma_{k+1}^P$, but these inclusions are not known to be strict. See Figure 1.

The classes Σ_k^P and Π_k^P constitute the **polynomial hierarchy**. Define

$$\mathsf{PH} = \bigcup_{k \geq 0} \Sigma_k^P .$$

It is straightforward to prove that $\mathsf{PH} \subseteq \mathsf{PSPACE}$, but it is not known whether the inclusion is strict. In fact, if $\mathsf{PH} = \mathsf{PSPACE}$, then the polynomial hierarchy collapses to some level, i.e., $\mathsf{PH} = \Sigma_m^P$ for some m.

We have already hinted that the levels of the polynomial hierarchy correspond to k-alternating Turing machines. The next theorem makes this correspondence explicit, and also gives us a third equivalent characterization.

Theorem 2.1 *For any language A, the following are equivalent:*

1. *$A \in \sum_k^P$.*

2. *A is decided in polynomial time by a k-alternating Turing machine that starts in an existential state.*

3. *There exists a language $B \in \mathsf{P}$ and a polynomial p such that for all x, $x \in A$ if and only if*

$$(\exists y_1 : |y_1| \leq p(|x|))(\forall y_2 : |y_2| \leq p(|x|)) \cdots (Q y_k : |y_k| \leq p(|x|))[(x, y_1, \ldots, y_k) \in B],$$

where the quantifier Q is $\exists$ if k is odd, $\forall$ if k is even.

In Section 8 of Chapter 28, we discussed some of the startling consequences that would follow if NP were included in $\mathsf{P/poly}$, but observed that this inclusion was not known to imply $\mathsf{P} = \mathsf{NP}$. It is known, however, that if $\mathsf{NP} \subseteq \mathsf{P/poly}$, then PH collapses to its second level, $\sum_2^P$ [Karp and Lipton, 1982]. It is generally considered likely that PH does not collapse to any level, and hence that all of its levels are distinct. Hence this result is considered strong evidence that NP is not a subset of $\mathsf{P/poly}$.

Also inside the polynomial hierarchy is the important class BPP of problems that can be solved efficiently and reliably by probabilistic algorithms, to which we now turn.

3 Probabilistic Complexity Classes

Since the 1970s, with the development of randomized algorithms for computational problems (see Chapter 15), complexity theorists have placed randomized algorithms on a firm intellectual foundation. In this section, we outline some basic concepts in this area.

A **probabilistic Turing machine** M can be formalized as a nondeterministic Turing machine with exactly two choices at each step. During a computation, M chooses each possible next step with independent probability $1/2$. Intuitively, at each step, M flips a fair coin to decide what to do next. The probability of a computation path of t steps is $1/2^t$. The probability that M accepts an input string x, denoted by $p_M(x)$, is the sum of the probabilities of the accepting computation paths.

Throughout this section, we consider only machines whose time complexity $t(n)$ is time-constructible. Without loss of generality, we may assume that every computation path of such a machine halts in exactly t steps.

Let A be a language. A probabilistic Turing machine M decides A with

		for all $x \in A$	for all $x \notin A$
unbounded two-sided error	if	$p_M(x) > 1/2$	$p_M(x) \leq 1/2$
bounded two-sided error	if	$p_M(x) > 1/2 + \epsilon$	$p_M(x) < 1/2 - \epsilon$
		for some constant ϵ	
one-sided error	if	$p_M(x) > 1/2$	$p_M(x) = 0$

Many practical and important probabilistic algorithms make one-sided errors. For example, in the Solovay-Strassen primality testing algorithm of Chapter 15 (on randomized algorithms), when the input x is a prime number, the algorithm *always* says "prime"; when x is composite, the algorithm *usually* says "composite," but may occasionally say "prime." Using the definitions above, this means that the Solovay-Strassen algorithm is a one-sided error algorithm for the set A of composite numbers. It also is a bounded two-sided error algorithm for $\overline{A}$, the set of prime numbers.

These three kinds of errors suggest three complexity classes:

- PP is the class of languages decided by probabilistic Turing machines of polynomial time complexity with unbounded two-sided error.

- BPP is the class of languages decided by probabilistic Turing machines of polynomial time complexity with bounded two-sided error.

- RP is the class of languages decided by probabilistic Turing machines of polynomial time complexity with one-sided error.

In the literature, RP is also called R.

A probabilistic Turing machine M is a PP-**machine** (respectively, a BPP-**machine**, an RP-**machine**) if M has polynomial time complexity, and M decides with two-sided error (bounded two-sided error, one-sided error).

Through repeated Bernoulli trials, we can make the error probabilities of BPP-machines and RP-machines arbitrarily small, as stated in the following theorem. (Among other things, this theorem implies that $\mathsf{RP} \subseteq \mathsf{BPP}$.)

Theorem 3.1 *If $L \in \mathsf{BPP}$, then for every polynomial $q(n)$, there exists a* BPP*-machine M such that $p_M(x) > 1 - 1/2^{q(n)}$ for every $x \in L$, and $p_M(x) < 1/2^{q(n)}$ for every $x \notin L$.*

If $L \in \mathsf{RP}$, then for every polynomial $q(n)$, there exists an RP*-machine M such that $p_M(x) > 1 - 1/2^{q(n)}$ for every x in L.*

It is important to note just how minuscule the probability of error is (provided that the coin flips are truly random). If the probability of error is less than $1/2^{5000}$, then it is less likely that the algorithm produces an incorrect answer than that the computer will be struck by a meteor. An algorithm whose probability of error is $1/2^{5000}$ is essentially as good as an algorithm that makes no errors. For this reason, many computer scientists consider BPP to be the class of practically feasible computational problems.

Next, we define a class of problems that have probabilistic algorithms that make no errors. Define

- $\mathsf{ZPP} = \mathsf{RP} \cap \text{co-}\mathsf{RP}$.

The letter Z in ZPP is for zero probability of error, as we now demonstrate. Suppose $A \in \mathsf{ZPP}$. Here is an algorithm that checks membership in A. Let M be an RP-machine that decides A, and let M' be an RP-machine that decides $\overline{A}$. For an input string x, alternately run M and M' on x, repeatedly, until a computation path of one machine accepts x. If M accepts x, then accept x; if M' accepts x, then reject x. This algorithm works correctly because when an RP-machine accepts its input, it does not make a mistake. This algorithm might not terminate, but with very high probability, the algorithm terminates after a few iterations.

The next theorem expresses some known relationships between probabilistic complexity classes and other complexity classes, such as classes in the polynomial hierarchy (see Section 2).

Theorem 3.2

(a) $\mathsf{P} \subseteq \mathsf{ZPP} \subseteq \mathsf{RP} \subseteq \mathsf{BPP} \subseteq \mathsf{PP} \subseteq \mathsf{PSPACE}$.

(b) $\mathsf{RP} \subseteq \mathsf{NP} \subseteq \mathsf{PP}$.

(c) $\mathsf{BPP} \subseteq \sum_2^P \cap \prod_2^P$.

(d) $\mathsf{PH} \subseteq \mathsf{P}^{\mathsf{PP}}$.

(e) $\mathsf{TC}^0 \subset \mathsf{PP}$.

(Note that the last inclusion is strict! TC^0 is not known to be different from NP, but it is a proper subset of PP.) Figure 2 illustrates many of these relationships. PP is not considered to be a feasible class because it contains NP.

Even though it is not clear that there is a good physical source of randomness that can be used to execute probabilistic algorithms and obtain the desired low error bounds, pseudo-random generators are often used and seem to work well. There is currently great interest in de-randomizing probabilistic algorithms, but that topic is beyond the scope of this chapter. There is a simple sense in which a probabilistic algorithm can be de-randomized, however. If an algorithm has very small error probability (in particular, if it has error probability a little less than $1/2^n$), then there is one sequence of coin flips that gives the right answer on all inputs of length n, and this sequence can be hard-wired into the algorithm to yield a deterministic (but *non-uniform*) circuit family. More formally:

Theorem 3.3 $\mathsf{BPP} \subseteq \mathsf{P/poly}$.

There is another important way in which $\mathsf{BPP}, \mathsf{RP}$, and ZPP differ from PP (as well as from NP and all of the other complexity classes we have discussed thus far): BPP, RP, and ZPP are not known to have any complete languages. Intuitively, BPP is believed to lack complete sets because there is no computable way to weed out those polynomial-time probabilistic Turing machines that are not BPP-machines from those that are. The same goes for RP and ZPP—a more detailed discussion of this point may be found in [Sipser, 1982] and [Ambos-Spies, 1986]. To be sure, if these classes equal P then trivially they have complete languages. Recent work [Impagliazzo and Wigderson, 1997] proves that a highly plausible hardness assertion for languages in exponential time implies $\mathsf{P} = \mathsf{BPP}$.

Log-space analogues of these probabilistic classes have also been studied, of which the most important is RL, defined by probabilistic TMs with one-sided error that run in log space and may

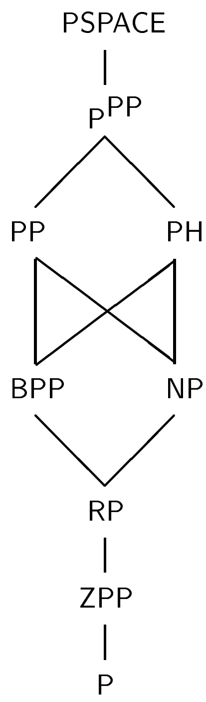

Figure 2: Probabilistic complexity classes.

use polynomially many random bits in any computation. An important problem in RL that is not known to be in L is that of whether there is a path from node s to node t in an *un*directed graph, or much the same thing, whether an undirected graph is connected.

4 Formal Logic and Complexity Classes

There is a surprisingly close connection between important complexity classes and natural notions that arise in the study of formal logic. This connection has led to important applications of complexity theory to logic, and vice-versa. Below, we present some basic notions from formal logic, and then we show some of the connections between logic and complexity theory.

Descriptive complexity refers to the ability to describe and characterize individual problems and whole complexity classes by certain kinds of formulas in formal logic. These descriptions do not depend on an underlying machine model—they are machine-independent. Furthermore, computational problems can be described in terms of their native data structures, rather than under *ad hoc* string encodings.

A **relational structure** consists of a set V (called the *universe*), a tuple $E_1, \ldots, E_k$ of relations on V, and a tuple $c_1, \ldots, c_\ell$ of elements of V ($k, \ell \geq 0$). Its *type* τ is given by the tuple $(a_1, \ldots, a_k)$ of arities of the respective relations, together with ℓ. In this chapter, V is always finite. For example, directed graphs $G = (V, E)$ are relational structures with the one binary relation E, and their type has $k = 1$, $a_1 = 2$, and $\ell = 0$, the last since there are no distinguished vertices. For another example, instances of the GRAPH ACCESSIBILITY PROBLEM (GAP) consist of a directed graph $G = (V, E)$ along with two distinguished vertices $s, t \in V$ (see Chapter 28, Section 5), so they have $\ell = 2$.

An ordinary binary string x can be regarded as a structure $(V, X, \leq)$, where $\leq$ is a total order on V that sequences the bits, and for all i ($1 \leq i \leq |x|$), $x_i = 1$ if and only if $X(u_i)$ holds. Here u_i is the ith element of V under the total order, and x_i is the ith bit of x. It is often desirable to regard the ordering $\leq$ as fixed, and focus attention on the single unary relation $X(\cdot)$ as the essence of the string.

4.1 Systems of Logic

For our purposes, a **system of logic** (or *logic language*) $\mathcal{L}$ consists of the following:

1. A tuple $(\mathbf{E}_1, \ldots, \mathbf{E}_k)$ of *relation symbols*, with corresponding arities $a_1, \ldots, a_k \geq 1$, and a tuple $(\mathbf{c}_1, \ldots, \mathbf{c}_\ell)$ of *constant symbols* ($k, \ell \geq 0$). These symbols constitute the *vocabulary* of $\mathcal{L}$, and can be identified with the corresponding type τ of relational structures.

2. Optionally, a further finite collection of relation and constant symbols whose interpretations are fixed in all universes V under consideration. By default this collection contains the symbol $=$, which is interpreted as the equality relation on V.

3. An unbounded supply of variable symbols $u, v, w, \ldots$ ranging over elements of V, and optionally, an unbounded supply of variable relation symbols $R_1, R_2, R_3, \ldots$, each with an associated arity and ranging over relations on V.

4. A complete set of Boolean connectives, for which we use $\wedge$, $\vee$, $\neg$, $\rightarrow$, and $\leftrightarrow$, and the quantifiers $\forall$, $\exists$. Additional kinds of operators for building up formulas are discussed later.

The *well-formed formulas* of $\mathcal{L}$, and the *free*, *bound*, *positive*, and *negative* occurrences of symbols in a formula, are defined in the usual inductive manner. A *sentence* is a formula ϕ with no free variables. A formula, or a whole system, is called **first-order** if it has no relation *variables* R_i; otherwise it is **second-order**.

Just as machines of a particular type define complexity classes, so also do logical formulas of a particular type define important classes of languages. The most common nomenclature for these classes begins with a prefix such as FO or F_1 for first-order systems, and SO or F_2 for second-order. $\mathsf{SO}\exists$ denotes systems whose second-order formulas are restricted to the form $(\exists R_1)(\exists R_2)\ldots(\exists R_k)\psi$ with ψ first-order. After this prefix, in parentheses, we list the vocabulary, and any extra fixed-interpretation symbols or additions to formulas. For instance, $\mathsf{SO}\exists(\mathit{Graphs}, \leq)$ stands for the second-order existential theory of graphs whose nodes are labeled and ordered. (The predicate $=$ is always available in the logics we study, and thus it is not explicitly listed with the other fixed-interpretation symbols such as $\leq$.)

The fixed-interpretation symbols deserve special mention. Many authorities treat them as part of the vocabulary. A finite universe V may without loss of generality be identified with the set $\{1, \ldots, n\}$, where $n \in \mathbf{N}$. Important fixed-interpretation symbols for these sets, besides $=$ and $\leq$, are Suc, $+$, and $*$, respectively standing for the successor, addition, and multiplication relations. (Here $+(i, j, k)$ stands for $i + j = k$, etc.) Insofar as they deal with the numeric coding of V and do not depend on any structures that are being built on V, such fixed-interpretation symbols are commonly called *numerical predicates*.

4.2 Languages, Logics, and Complexity Classes

Let us see how a logical formula describes a language, just as a Turing machine or a program does. A formal inductive definition of the following key notion, and much further information on systems of logic, may be found in the standard text [Enderton, 1972].

Definition 4.1. Let ϕ be a sentence in a system $\mathcal{L}$ with vocabulary τ. A relational structure $\mathcal{R}$ of type τ *satisfies* (or *models*) ϕ, written $\mathcal{R} \models \phi$, if ϕ becomes a true statement about $\mathcal{R}$ when the elements of $\mathcal{R}$ are substituted for the corresponding vocabulary symbols of ϕ. The **language of** ϕ is $L_\phi = \{\mathcal{R} : \mathcal{R} \models \phi\}$.

We say that ϕ *describes* L_ϕ, or describes the property of belonging to L_ϕ. Finally, given a system $\mathcal{L}$ of vocabulary τ, $\mathcal{L}$ itself stands for the class of structures of type τ that are described by formulas in $\mathcal{L}$. If τ is the vocabulary *Strings* of binary strings, then L_ϕ is a language in the familiar sense of a subset of $\{0,1\}^*$, and systems $\mathcal{L}$ over τ define ordinary classes of languages. Thus defining sets of structures over τ generalizes the notion of defining languages over an alphabet.

For example, the formula $(\forall u)X(u)$ over binary strings describes the language 1^*, while $(\forall v,w)[v \neq w \leftrightarrow E(v,w)]$ defines complete (loop-free) graphs. The formula

$$\mathit{Undir} \;=\; (\forall v,w)[E(v,w) \rightarrow E(w,v)] \wedge (\forall u)\neg E(u,u)$$

describes the property of being an undirected simple graph, treating an undirected edge as a pair of directed edges, and ruling out "self-loops." Given unary relation symbols $X_1,\ldots,X_k$, the formula

$$\mathit{Uniq}_{X_1,\ldots,X_k} \;=\; (\forall v)[\bigvee_{1\leq i\leq k} X_i(v) \wedge \bigwedge_{1\leq i<j\leq k} \neg(X_i(v) \wedge X_j(v))]$$

expresses that every element v is assigned exactly one i such that $X_i(v)$ holds. Given an arbitrary finite alphabet $\Sigma = \{c_1,\ldots,c_k\}$, the vocabulary $\{X_1,\ldots,X_k\}$, together with this formula, enables us to define languages of strings over Σ. (Since the presence of *Uniq* does not affect any of the syntactic characterizations that follow, we may now regard *Strings* as a vocabulary over any Σ.) Given a unary relation symbol R and the numerical predicate *Suc* on V, the formula

$$\mathit{Alts}_R \;=\; (\exists s,t)(\forall u,v)[\neg \mathit{Suc}(u,s) \wedge \neg \mathit{Suc}(t,u) \wedge R(s) \wedge \neg R(t) \wedge (\mathit{Suc}(u,v) \rightarrow (R(u) \leftrightarrow \neg R(v))]$$

says that R is true of the first element s, false of the last element t, and alternates true and false in-between. This requires $|V|$ to be even. The following examples are used again below.

(1) The regular language $(10)^*$ is described by the first-order formula $\phi_1 \;=\; \mathit{Alts}_X$.

(2) $(11)^*$ is described by the second-order formula $\phi_2 \;=\; (\exists R)(\forall u)[X(u) \wedge \mathit{Alts}_R]$.

(3) Graph Three-Colorability:

$$\phi_3 \;=\; \mathit{Undir} \wedge (\exists R_1,R_2,R_3)[\mathit{Uniq}_{R_1,R_2,R_3} \wedge (\forall v,w)(E(v,w) \rightarrow \bigvee_{1\leq i\leq 3} R_i(v) \wedge \neg R_i(w))].$$

(4) GAP (i.e., s-t connectivity for directed graphs):

$$\phi_4 \;=\; (\forall R)\neg(\forall u,v)[R(s) \wedge \neg R(t) \wedge (R(u) \wedge E(u,v) \rightarrow R(v))].$$

Formula ϕ_4 says that there is no set $R \subseteq V$ that is closed under the edge relation and contains s but doesn't contain t, and this is equivalent to the existence of a path from s to t. Much trickier is the fact that deleting "$Uniq_{R_1,R_2,R_3}$" from ϕ_3 leaves a formula that still defines exactly the set of undirected 3-colorable graphs. This fact hints at the delicacy of complexity issues in logic.

Much of this study originated in research on database systems, because data base query languages correspond to logics. First-order logic is notoriously limited in expressive power, and this limitation has motivated the study of extensions of first-order logic, such as the following *first-order operators*.

Definition 4.2.

(a) *Transitive closure* (TC): Let ϕ be a formula in which the first-order variables $u_1, \ldots, u_k$ and $v_1, \ldots, v_k$ occur freely, and regard ϕ as implicitly defining a binary relation S on V^k. That is, S is the set of pairs $(\vec{u}, \vec{v})$ such that $\phi(\vec{u}, \vec{v})$ holds. Then $\mathrm{TC}_{(u_1,\ldots,u_k,v_1,\ldots,v_k)}\,\phi$ is a formula, and its semantics is the reflexive-transitive closure of S.

(b) *Least fixed point* (LFP): Let ϕ be a formula with free first-order variables $u_1, \ldots, u_k$ and a free k-ary relation symbol R that occurs only positively in ϕ. In this case, for any relational structure $\mathcal{R}$ and $S \subseteq V^k$, the mapping $f_\phi(S) = \{\, (e_1, \ldots, e_k) : \mathcal{R} \models \phi(S, e_1, \ldots, e_k) \,\}$ is monotone. That is, if $S \subseteq T$, then for every tuple of domain elements $(e_1, \ldots, e_k)$, if $\phi(R, u_1, \ldots, u_k)$ evaluates to `true` when R is set to S and each u_i is set to e_i, then ϕ also evaluates to `true` when R is set to T, because R appears positively. Thus the mapping f_ϕ has a least fixed point in V^k. Then $\mathrm{LFP}_{(R,u_1,\ldots,u_k)}\,\phi$ is a formula, and its semantics is the least fixed point of f_ϕ, i.e., the smallest S such that $f_\phi(S) = S$.

(c) *Partial fixed point* (PFP): Even if f_ϕ above is not monotone, $\mathrm{PFP}_{(R,u_1,\ldots,u_k)}\,\phi$ is a formula whose semantics is the first fixed point found in the sequence $f_\phi(\emptyset), f_\phi(f_\phi(\emptyset)), \ldots$, if it exists, $\emptyset$ otherwise.

The first-order variables $u_1, \ldots, u_k$ remain free in these formulas. The relation symbol R is bound in $\mathrm{LFP}_{(R,u_1,\ldots,u_k)}\,\phi$, but since this formula is fixing R uniquely rather than quantifying over it, the formula $\mathrm{LFP}_{(R,u_1,\ldots,u_k)}\,\phi$ is still regarded as first-order (provided ϕ is first-order).

A somewhat less natural but still useful operation is the "deterministic transitive closure" operator. We write "DTC" for the restriction of (a) above to cases where the implicitly defined binary relation S is a partial function. The DTC restriction is enforcible syntactically by replacing any (sub)-formula ϕ to which TC is applied by $\phi'' = \phi \wedge (\forall w_1, \ldots, w_k)[\phi' \rightarrow \wedge_{i=1}^{k} v_i = w_i]$, where ϕ' is the result of replacing each v_i in ϕ by w_i, $1 \leq i \leq k$.

For example, s-t connectivity is definable by the FO(TC) and FO(LFP) formulas

$$\begin{aligned} \phi_4' &= (\exists u, v)[u = s \wedge v = t \wedge \mathrm{TC}_{(u,v)}\, E(u,v)], \\ \phi_4'' &= (\exists u, v)[u = s \wedge v = t \wedge \mathrm{LFP}_{(R,u,v)}\, \psi], \end{aligned}$$

where $\psi = (u = v \vee E(u,v) \vee (\exists w)[R(u,w) \wedge R(w,v)])$. To understand how ϕ_4'' works, starting with S as the empty binary relation and substituting the current S for R at each turn, the first iteration yields $S = \{\, (u,v) : u = v \vee E(u,v) \,\}$, the second iteration gives pairs of vertices connected by a path of length at most 2, then 4, ..., and the fixed-point is the reflexive-transitive closure E^* of E. Then ϕ_4'' is read as if it were $(\exists u, v)(u = s \wedge v = t \wedge E^*(u,v))$, or more simply, as if it were $E^*(s,t)$.

Note however, that writing DTC... in place of TC... in ϕ_4' changes the property defined by restricting it to directed graphs in which each non-sink vertex has out-degree 1. It is not known whether s-t connectivity can be expressed using the DTC operator. This question is equivalent to whether $\mathsf{L} = \mathsf{NL}$.

4.3 Logical Characterizations of Complexity Classes

As discussed by [Fagin, 1993], there is a uniform encoding method *Enc* such that for any vocabulary τ and (finite) relational structure $\mathcal{R}$ of type τ, $\mathit{Enc}(\mathcal{R})$ is a standard string encoding of $\mathcal{R}$. For instance with $\tau = \mathit{Graphs}$, an n-vertex graph becomes the size-n^2 binary string that lists the entries of its adjacency matrix in row-major order. Thus one can say that a language L_ϕ over any vocabulary belongs to a complexity class $\mathcal{C}$ if the string language $\mathit{Enc}(L_\phi) = \{\, \mathit{Enc}(\mathcal{R}) : \mathcal{R} \models \phi \,\}$ is in $\mathcal{C}$.

The following theorems of the form "$\mathcal{C} = \mathcal{L}$" all hold in the following strong sense: for every vocabulary τ and $\mathcal{L}(\tau)$-formula ϕ, $\mathit{Enc}(L_\phi) \in \mathcal{C}$; and for every language $A \in \mathcal{C}$, there is a $\mathcal{L}(\mathit{Strings})$-formula ϕ such that $L_\phi = A$. Although going to strings via *Enc* may seem counter to the motivation expressed in the first paragraph of this whole section, the generality and strength of these results has a powerful impact in the desired direction: they define the right notion of complexity class $\mathcal{C}$ for any vocabulary τ. Hence we omit the vocabulary τ in the following statements.

Theorem 4.1

(a) $\mathsf{PSPACE} = \mathsf{FO}(\mathrm{PFP}, \leq)$.

(b) $\mathsf{PH} = \mathsf{SO}$.

(c) **(Fagin's Theorem)** $\mathsf{NP} = \mathsf{SO}\exists$.

(d) $\mathsf{P} = \mathsf{FO}(\mathrm{LFP}, \leq)$.

(e) $\mathsf{NL} = \mathsf{FO}(\mathrm{TC}, \leq)$.

(f) $\mathsf{L} = \mathsf{FO}(\mathrm{DTC}, \leq)$.

(g) $\mathsf{AC}^0 = \mathsf{FO}(+, *)$.

One other result should be mentioned with the above. Define the *spectrum* of a formula ϕ by $S_\phi = \{n : \text{ for some } \mathcal{R} \text{ with } n \text{ elements, } \mathcal{R} \models \phi\}$. The paper [Jones and Selman, 1974] proved that a language A belongs to NE if and only if there is a vocabulary τ and a sentence $\phi \in \mathsf{FO}(\tau)$ such that $A = S_\phi$ (identifying numbers and strings). Thus spectra characterize NE.

The ordering $\leq$ is needed in results (a), (d), (e), and (f). The paper [Chandra and Harel, 1982] proved that $\mathsf{FO}(\mathrm{LFP})$ without $\leq$ cannot even define $(11)^*$ (and their proof works also for $\mathsf{FO}(\mathrm{PFP})$). Put another way, without an (ad-hoc) ordering on the full database, one cannot express queries of the kind "Is the number of widgets in Toledo even?" even in the powerful system of first-order logic with PFP. Note that, as a consequence of what we know about complexity classes, it follows that $\mathsf{FO}(\mathrm{PFP}, \leq)$ is more expressive than $\mathsf{FO}(\mathrm{TC}, \leq)$. This result is an example of an application of complexity theory to logic. In contrast, when the ordering is not present, it is much easier to show directly that $\mathsf{FO}(\mathrm{PFP})$ is more powerful than $\mathsf{FO}(\mathrm{TC})$ than to use the tools of complexity theory. Furthermore, the hypotheses $\mathsf{FO}(\mathrm{LFP}) = \mathsf{FO}(\mathrm{PFP})$ and $\mathsf{FO}(\mathrm{LFP}, \leq) = \mathsf{FO}(\mathrm{PFP}, \leq)$ are *both* equivalent to $\mathsf{P} = \mathsf{PSPACE}$ [Abiteboul and Vianu, 1995]. This shows how logic can apply to complexity theory.

4.4 A Short Digression: Logic and Formal Languages

There are two more logical characterizations that seem at first to have little to do with complexity theory. Characterizations such as these have been important in circuit complexity, but those considerations are beyond the scope of this chapter.

Let SF stand for the class of *star-free regular languages,* which are defined by regular expressions without Kleene stars, but with $\emptyset$ as an atom and complementation ($\sim$) as an operator. For example, $(10)^* \in \mathsf{SF}$ via the equivalent expression $\sim [(\sim \emptyset)(00+11)(\sim \emptyset) + 0(\sim \emptyset) + (\sim \emptyset)1]$.

A formula is *monadic* if each of its relation symbols is unary. A second-order system is *monadic* if every relation variable is unary. Let mSO denote the monadic second-order formulas. The formula ϕ_2 above defines $(11)^*$ in $\mathsf{mSO}\exists(Suc)$. The following results are specific to the vocabulary of strings.

Theorem 4.2

(a) $\mathsf{REG} = \mathsf{mSO}(Strings, \leq) = \mathsf{mSO}\exists(Strings, Suc)$.

(b) $\mathsf{SF} = \mathsf{FO}(Strings, \leq)$.

Theorem 4.2, combined with Theorem 4.1 (b) and (c), shows that SO is much more expressive than mSO, and $\mathsf{SO}\exists(\leq)$ is similarly more expressive than $\mathsf{mSO}\exists(\leq)$. A seemingly smaller change to $\mathsf{mSO}\exists$ also results in a leap of expressiveness from the regular languages to the level of NP. The paper [Lynch, 1982] showed that if we consider $\mathsf{mSO}\exists(+)$ instead of $\mathsf{mSO}\exists(\leq)$ (for strings), then the resulting class contains $\mathsf{NTIME}[n]$, and hence contains many NP-complete languages, such as Graph Three-Colorability.

5 Interactive Models and Complexity Classes

5.1 Interactive Proofs

In Section 2.2 of Chapter 27, we characterized NP as the set of languages whose membership proofs can be checked quickly, by a deterministic Turing machine M of polynomial time complexity. A different notion of proof involves interaction between two parties, a prover P and a verifier V, who exchange messages. In an **interactive proof system**, the prover is an all-powerful machine, with unlimited computational resources, analogous to a teacher. The verifier is a computationally limited machine, analogous to a student. Interactive proof systems are also called "Arthur-Merlin games": the wizard Merlin corresponds to P, and the impatient Arthur corresponds to V.

Formally, an **interactive proof system** comprises the following:

- A read-only input tape on which an input string x is written.
- A *prover* P, whose behavior is not restricted.
- A *verifier* V, which is a probabilistic Turing machine augmented with the capability to send and receive messages. The running time of V is bounded by a polynomial in $|x|$.
- A tape on which V writes messages to send to P, and a tape on which P writes messages to send to V. The length of every message is bounded by a polynomial in $|x|$.

A computation of an interactive proof system (P, V) proceeds in rounds, as follows. For $j = 1, 2, \ldots$, in round j, V performs some steps, writes a message m_j, and temporarily stops. Then P reads m_j and responds with a message m'_j, which V reads in round $j+1$. An interactive proof system (P, V) **accepts** an input string x if the probability of acceptance by V satisfies $p_V(x) > 1/2$.

In an interactive proof system, a prover can convince the verifier about the truth of a statement without exhibiting an entire proof, as the following example illustrates.

Example: Consider the graph non-isomorphism problem: the input consists of two graphs G and H, and the decision is "yes" if and only if G is not isomorphic to H. Although there is a short proof that two graphs *are* isomorphic (namely: the proof consists of the isomorphism mapping G onto H), nobody has found a general way of proving that two graphs are *not* isomorphic that is significantly shorter than listing all $n!$ permutations and showing that each fails to be an isomorphism. (That is, the graph non-isomorphism problem is in co-NP, but is not known to be in NP.) In contrast, the verifier V in an interactive proof system is able to take statistical evidence into account, and determine "beyond all reasonable doubt" that two graphs are non-isomorphic, using the following protocol.

In each round, V randomly chooses either G or H with equal probability; if V chooses G, then V computes a random permutation G' of G, presents G' to P, and asks P whether G' came from G or from H (and similarly if V chooses H). If P gave an erroneous answer on the first round, and G is isomorphic to H, then after k subsequent rounds, the probability that P answers all the subsequent queries correctly is $1/2^k$. (To see this, it is important to understand that the prover P does not see the coins that V flips in making its random choices; P sees only the graphs G' and H' that V sends as messages.) V accepts the interaction with P as "proof" that G and H are non-isomorphic if P is able to pick the correct graph for 100 consecutive rounds. Note that V has ample grounds to accept this as a convincing demonstration: if the graphs are indeed isomorphic, the prover P would have to have an incredible streak of luck to fool V.

The complexity class IP comprises the languages A for which there exists a verifier V and an ϵ such that

- there exists a prover $\hat{P}$ such that for all x in A, the interactive proof system $(\hat{P}, V)$ accepts x with probability greater than $1/2 + \epsilon$; and
- for every prover P and every $x \notin A$, the interactive proof system (P, V) rejects x with probability greater than $1/2 + \epsilon$.

By substituting random choices for existential choices in the proof that $\mathsf{ATIME}(t) \subseteq \mathsf{DSPACE}(t)$ (Theorem 2.8 in Chapter 27), it is straightforward to show that $\mathsf{IP} \subseteq \mathsf{PSPACE}$. It was originally believed likely that IP was a small subclass of PSPACE. Evidence supporting this belief was the construction in [Fortnow and Sipser, 1988] of an oracle language B for which co-$\mathsf{NP}^B - \mathsf{IP}^B \neq \emptyset$, so that IP^B is strictly included in PSPACE^B. Using a proof technique that does not relativize, however, [Shamir, 1992] (building on the work of [Lund *et al.*, 1992]) proved that in fact, IP and PSPACE are the same class.

Theorem 5.1 $\mathsf{IP} = \mathsf{PSPACE}$.

If NP is a proper subset of PSPACE, as is widely believed, then Theorem 5.1 says that interactive proof systems can decide a larger class of languages than NP.

5.2 Probabilistically Checkable Proofs

In an interactive proof system, the verifier does not need a complete conventional proof to become convinced about the membership of a word in a language, but uses random choices to query parts of a proof that the prover may know. This interpretation inspired another notion of "proof": a proof consists of a (potentially) large amount of information that the verifier need only inspect in a few places in order to become convinced. The following definition makes this idea more precise.

A language L has a **probabilistically checkable proof** if there exists an oracle BPP-machine M such that

- for all $x \in L$, there exists an oracle language B_x such that M^{B_x} accepts x.
- for all $x \notin L$, and for every language B, machine M^B rejects x.

Intuitively, the oracle language B_x represents a proof of membership of x in L. Notice that B_x can be finite since the length of each possible query during a computation of M^{B_x} on x is bounded by the running time of M. The oracle language takes the role of the prover in an interactive proof system—but in contrast to an interactive proof system, the prover cannot change strategy adaptively in response to the questions that the verifier poses. This change results in a potentially stronger system, since a machine M that has bounded error probability relative to all languages B might not have bounded error probability relative to some adaptive prover. Although this change to the proof system framework may seem modest, it leads to a characterization of a class that seems to be much larger than PSPACE.

Theorem 5.2 *A has a probabilistically checkable proof if and only if $A \in$ NEXP.*

Although the notion of probabilistically checkable proofs seems to lead us away from feasible complexity classes, by considering natural restrictions on how the proof is accessed, we can obtain important insights into familiar complexity classes.

Let $\mathsf{PCP}(r(n), q(n))$ denote the class of languages with probabilistically checkable proofs in which the probabilistic oracle Turing machine M makes $O(r(n))$ random binary choices, and queries its oracle $O(q(n))$ times. (For this definition, we assume that M has either one or two choices for each step.) It follows from the definitions that $\mathsf{BPP} = \mathsf{PCP}(n^{O(1)}, 0)$, and $\mathsf{NP} = \mathsf{PCP}(0, n^{O(1)})$.

Theorem 5.3 $\mathsf{NP} = \mathsf{PCP}(\log n, 1)$.

Theorem 5.3 asserts that for every language L in NP, a proof that $x \in L$ can be encoded so that the verifier can be convinced of the correctness of the proof (or detect an incorrect proof) by using only $O(\log n)$ random choices, and inspecting only a *constant* number of bits of the proof!

This surprising characterization of NP has important applications to the complexity of finding approximate solutions to optimization problems, as discussed in Section 6.2 below.

6 Classifying the Complexity of Functions

Up to now, we have considered only the complexity of decision problems. (Recall that a decision problem is a problem in which, for every input, the output is either "yes" or "no".) Most of the functions that we actually compute are functions that produce more than one bit of output. For example, instead of merely deciding whether a graph has a clique of size m, we often want to *find* a clique. Problems in NP are naturally associated with this kind of search problem.

Of course, any function f can be analyzed in terms of a decision problem in a straightforward way by considering the decision problem A_f that takes as input x and i, and answers "yes" if the i^{th} bit of $f(x)$ is 1. But there are other ways of formulating functions as decision problems, and sometimes it is instructive to study the complexity of functions directly instead of their associated decision problems. In this section and the sections that follow, we will discuss some of the more useful classifications.

The most important class of functions is the class that we can compute quickly.

- FP is the set of functions computable in polynomial time by deterministic Turing machines.

In an analogous way, we define FL, FNC^k, etc., to be the set of functions computable by deterministic log-space machines, by NC^k circuits, etc. We also define $\mathsf{FPSPACE}$ to be the class of functions f computable by deterministic machines in polynomial space, such that also $|f(x)|$ is bounded by a polynomial in $|x|$. This restriction is essential because a machine that uses polynomial space could run for an exponential number of steps, producing an exponentially long output.

To study functions that appear to be difficult to compute, we again use the notions of reducibility and completeness. Analogous to Cook reducibility to oracle *languages*, we consider Cook reducibility to a *function* given as an oracle. For a function f whose length $|f(x)|$ is bounded by a polynomial in $|x|$, we say that a language A is Cook reducible to f if there is a polynomial-time oracle Turing machine M that accepts A, where the oracle is accessed as follows: M writes a string y on the query tape, and in the next step y is replaced by $f(y)$. As usual, we let P^f and FP^f denote the class of languages and functions computable in polynomial time with oracle f, respectively.

Let $\mathcal{C}$ be a class of functions. When $\mathcal{C}$ is at least as big as FP, then we will use Cook reducibility to define completeness. That is, a function f is $\mathcal{C}$-complete, if f is in $\mathcal{C}$ and $\mathcal{C} \subseteq \mathsf{FP}^f$. When we are discussing smaller classes $\mathcal{C}$ (where polynomial-time is too powerful to give a meaningful notion of reducibility), then when we say that a function f is $\mathcal{C}$-complete, it refers to completeness under AC^0-Turing reducibility, which was defined in Chapter 28, Section 6. In this chapter, we consider only these two variants of Turing reducibility. There are many other ways to reduce one function to another, just as there are many kinds of reductions between languages.

We use these notions to study optimization problems in Sections 6.1 and 6.2 and counting problems in Section 7.

6.1 Optimization Classes

Given an optimization (minimization) problem, we most often study the following associated decision problem:

> "Is the optimal value at most k"?

Alternatively, we could formulate the decision problem as the following:

> "Is the optimal value exactly k?"

For example, consider the TRAVELING SALESPERSON problem (TSP) again. TSP asks whether the length of the optimal tour is at most d_0. Define EXACT TSP to be the decision problem that asks whether the length of the optimal tour is exactly d_0. It is not clear that EXACT TSP is in NP or in $\mathsf{co\text{-}NP}$, but EXACT TSP *can* be expressed as the intersection of TSP and its complement $\overline{\text{TSP}}$: the length of the optimal tour is d_0 if there is a tour whose length is at most d_0, and no tour whose length is at most $d_0 - 1$. Similar remarks apply to the optimization problem MAX CLIQUE: given an undirected graph G, find the maximum size of a clique in G.

Exact versions of many optimization problems can be expressed as the intersection of a language in NP and a language in $\mathsf{co\text{-}NP}$. This observation motivates the definition of a new complexity class:

- D^P is the class of languages A such that $A = A_1 \cap A_2$ for some languages A_1 in NP and A_2 in co-NP.

The letter D in D^P means difference: $A \in \mathsf{D}^P$ if and only if A is the difference of two languages, i.e., $A = A_1 - A_3$ for some languages A_1 and A_3 in NP.

Not only is EXACT TSP in D^P, but also EXACT TSP is D^P-complete. Exact versions of many other NP-complete problems, including CLIQUE, are also D^P-complete [Papadimitriou and Yannakakis, 1984].

Although it is not known whether D^P is contained in NP, it is straightforward to prove that

$$\mathsf{NP} \subseteq \mathsf{D}^P \subseteq \mathsf{P}^{\mathsf{NP}} \subseteq \sum_2^P \cap \prod_2^P.$$

Thus, D^P lies between the first two levels of the polynomial hierarchy.

We have characterized the complexity of computing the optimal value of an instance of an optimization problem, but we have not yet characterized the complexity of computing the optimal solution itself. An optimization algorithm produces not only a "yes" or "no" answer, but also, when feasible solutions exist, an optimal solution.

First, for a maximization problem, suppose that we have a subroutine that solves the decision problem "Is the optimal value at least k?" Sometimes, with repeated calls to the subroutine, we can construct an optimal solution. For example, suppose subroutine S solves the CLIQUE problem; for an input graph G and integer k, the subroutine outputs "yes" if G has a clique of k (or more) vertices. To construct the largest clique in an input graph, first, determine the size K of the largest clique by binary search on $\{1, \ldots, n\}$ with $\log_2 n$ calls to S. Next, for each vertex v, in sequence, determine whether deleting v produces a graph whose largest clique has size K by calling S. If so, then delete v and continue with the remaining graph. If not, then look for a clique of size $K - 1$ among the neighbors of v.

The method outlined in the last paragraph uses S in the same way as an oracle Turing machine queries an oracle language in NP. With this observation, we define the following classes:

- $\mathsf{FP}^{\mathsf{NP}}$ is the set of functions computable in polynomial time by deterministic oracle Turing machines with oracle languages in NP.

- $\mathsf{FP}^{\mathsf{NP}[\log n]}$ is the set of functions computable in polynomial time by deterministic oracle Turing machines with oracle languages in NP that make $O(\log n)$ queries during computations on inputs of length n

$\mathsf{FP}^{\mathsf{NP}}$ and $\mathsf{FP}^{\mathsf{NP}[\log n]}$ contain many well-studied optimization problems (see [Krentel, 1988]). The problem of producing the optimal tour in the TRAVELING SALESPERSON problem is $\mathsf{FP}^{\mathsf{NP}}$-

complete. The problem of determining the size of the largest clique subgraph in a graph is $\mathsf{FP}^{\mathsf{NP}[\log n]}$-complete.

6.2 Approximability and Complexity

As discussed in Chapter 34 (on approximation algorithms), because polynomial-time algorithms for NP-complete optimization problems are unlikely to exist, we ask whether a polynomial-time algorithm can produce a feasible solution that is *close* to optimal.

Fix an optimization problem Π with a positive integer-valued objective function g. For each problem instance x, let $\mathsf{OPT}(x)$ be the optimal value, that is, $g(z)$, where z is a feasible solution to x that achieves the best possible value of g. Let M be a deterministic Turing machine that on input x produces as output a feasible solution $M(x)$ for Π. We say M is an **ϵ-approximation algorithm** if for all x,

$$\frac{|g(M(x)) - \mathsf{OPT}(x)|}{\max\{g(M(x)), \mathsf{OPT}(x)\}} \leq \epsilon.$$

(This definition handles both minimization and maximization problems.) The problem Π has a **polynomial-time approximation scheme** if for every fixed ϵ, there is a polynomial-time ϵ-approximation algorithm. Although the running time is polynomial in $|x|$, the time could be exponential in $1/\epsilon$.

Several NP-complete problems, including KNAPSACK, have polynomial-time approximation schemes. It is natural to ask whether *all* NP-complete optimization problems have polynomial-time approximation schemes. We define an important class of optimization problems, MAX-SNP, whose complete problems apparently do not.

First, we define a reducibility between optimization problems that preserves the quality of solutions. Let Π_1 and Π_2 be optimization problems with objective functions g_1 and g_2, respectively. An **L-reduction** from Π_1 to Π_2 is defined by a pair of polynomial-time computable functions f and f' that satisfy the following properties:

1. if x is an instance of Π_1 with optimal value $\mathsf{OPT}(x)$, then $f(x)$ is an instance of Π_2 whose optimal value satisfies $\mathsf{OPT}(f(x)) \leq c \cdot \mathsf{OPT}(x)$ for some constant c

2. if z is a feasible solution of $f(x)$, then $f'(z)$ is a feasible solution of x, such that

$$|\mathsf{OPT}(x) - g_1(f'(z))| \leq c'|\mathsf{OPT}(f(x)) - g_2(z)|$$

for some constant c'.

The second property implies that if z is an optimal solution to $f(x)$, then $f'(z)$ is an optimal solution to x. From the definitions, it follows that if there is an L-reduction from Π_1 to Π_2, and there is a polynomial-time approximation scheme for Π_2, then there is a polynomial-time approximation scheme for Π_1.

To define MAX-SNP, it will help to recall the characterization of NP as SO$\exists$ in Section 4.3. This characterization says that for any A in NP, there is a first-order formula ψ such that $x \in A$ if and only if

$$\exists S_1 \ldots \exists S_l \psi(x, S_1, \ldots, S_l).$$

For many important NP-complete problems, it is sufficient to consider having only a single second-order variable S, and to consider formulas ψ having only universal quantifiers. Thus, for such a language A we have a quantifier-free formula ϕ such that $x \in A$ if and only if

$$\exists S \forall u_1 \ldots \forall u_k \ \phi(S, u_1, \ldots, u_k).$$

Now define MAX-SNP$_0$ to be the class of optimization problems mapping input x to

$$\max_S |\{(y_1, \ldots, y_k) : \phi(S, y_1, \ldots, y_k)\}|.$$

For example, we can express in this form the MAX CUT problem, the problem of finding the largest cut in an input graph $G = (V, E)$ with vertex set V and edge set E. A set of vertices S is the optimal solution if it maximizes

$$|\{(v, w) : E(v, w) \wedge S(v) \wedge \neg S(w)\}|.$$

That is, the optimal solution S maximizes the number of edges (v, w) between vertices v in S and vertices w in $V - S$.

Define MAX-SNP to be the class of all optimization problems that are L-reducible to a problem in MAX-SNP$_0$. MAX-SNP contains many natural optimization problems. MAX CUT is MAX-SNP-complete, and MAX CLIQUE is MAX-SNP-hard, under L-reductions.

A surprising connection between the existence of probabilistically checkable proofs (Section 5.2) and the existence of approximation algorithms comes out in the next major theorem.

Theorem 6.1 *If there is a polynomial-time approximation scheme for some* MAX-SNP*-hard problem, then* P = NP*.*

In particular, unless P = NP, there is no polynomial-time approximation scheme for Max Cut or Max Clique. To prove this theorem, all we need to do is show its statement for a particular problem that is MAX-SNP-complete under L-reductions. However, we prefer to show the *idea* of the proof for the Max Clique problem, which although MAX-SNP-hard is not known to belong to MAX-SNP. It gives a strikingly different kind of reduction from an arbitrary language A in NP to Clique over the reduction from A to SAT to Clique in Chapter 28, Section 4, and its discovery by [Feige *et al.*, 1991, Feige *et al.*, 1996] stimulated the whole area.

Proof. Let $A \in \mathsf{NP}$. By Theorem 5.3, namely $\mathsf{NP} = \mathsf{PCP}[O(\log n), O(1)]$, there is a probabilistic oracle Turing machine M constrained to use $r(n) = O(\log n)$ random bits and make at most a constant number ℓ of queries in any computation path, such that:

- for all $x \in A$, there exists an oracle language B_x such that $\mathrm{Prob}_{s \in \{0,1\}^{r(n)}}[M^{B_x}(x,s) = 1] > 3/4$;
- for all $x \notin A$, and for every language B, $\mathrm{Prob}_{s \in \{0,1\}^{r(n)}}[M^{B}(x,s) = 1] < 1/4$.

Now define a *transcript* of M on input x to consist of a string $s \in \{0,1\}^{r(n)}$ together with a sequence of ℓ pairs (w_i, a_i), where w_i is an oracle query and $a_i \in \{0,1\}$ is a possible yes/no answer. In addition, a transcript must be *valid*: for all i, $0 \leq i < \ell$, on input x with random bits s, having made queries $w_1, \ldots, w_i$ to its (unspecified) oracle and received answers $a_1, \ldots, a_i$, machine M writes w_{i+1} as its next query string. Thus a transcript provides enough information to determine a full computation path of M on input x, and the transcript is *accepting* if and only if this computation path accepts. Finally, call two transcripts *consistent* if whenever a string w appears as "w_i" in one transcript and "w_j" in the other, the corresponding answer bits a_i and a_j are the same.

Construction: Let G_x be the undirected graph whose node set V_x is the set of all accepting transcripts, and whose edges connect pairs of transcripts that are consistent.

Complexity: Since $r(n) + \ell = O(\log n)$, there are only polynomially many transcripts, and since consistency is easy to check, G_x is constructed in polynomial time.

Correctness: If $x \in A$, then take the oracle B_x specified above and let C be the set of accepting transcripts whose answer bits are given by B_x. These transcripts are consistent with each other, and there are at least $(3/4)2^{r(n)}$ such accepting transcripts, so C forms a clique of size at least $(3/4)2^{r(n)}$ in G_x. Now suppose $x \notin A$, and suppose C' is a clique of size greater than $(1/4)2^{r(n)}$ in G_x. Because the transcripts in C' are mutually consistent, there exists a single oracle B that produces all the answer bits to queries in transcripts in C'. But then $\text{Prob}_s[M^B(x,s) = 1] > 1/4$, contradicting the PCP condition on M.

Thus we have proved the statement of the theorem for MAX CLIQUE. The proof of the general statement is similar. □

Note that the cases $x \in A$ and $x \notin A$ in this proof lead to a "(3/4,1/4) gap" in the maximum clique size ω of G_x. If there were a polynomial-time algorithm guaranteed to determine ω within a factor better than 3, then this algorithm could tell the "3/4" case apart from the "1/4" case, and hence decide whether $x \in A$. Since G_x can be constructed in polynomial time (in particular, G_x has size at most $2^{r(n)+\ell} = n^{O(1)}$), $\mathsf{P} = \mathsf{NP}$ would follow. Hence we can say that CLIQUE is NP-*hard to approximate within a factor of 3*. A long sequence of improvements to this basic construction has pushed the hardness-of-approximation not only to any fixed constant factor, but also to factors that increase with n. Moreover, approximation-preserving reductions have extended this kind of hardness result to many other optimization problems.

7 Counting

Two other important classes of functions deserve special mention:

- #P is the class of functions f such that there exists a nondeterministic polynomial-time Turing machine M with the property that $f(x)$ is the number of accepting computation paths of M on input x.

- #L is the class of functions f such that there exists a nondeterministic log-space Turing machine M with the property that $f(x)$ is the number of accepting computation paths of M on input x.

Some functions in $\#\mathsf{P}$ are clearly at least as difficult to compute as some NP-complete problems are to decide. For instance, consider the following problem.

Number of Satisfying Assignments to a 3CNF Formula (#3CNF)

Instance: A Boolean formula in conjunctive normal form with at most three variables per clause.

Output: The number of distinct assignments to the variables that cause the formula to evaluate to `true`.

Note that #3CNF is in $\#\mathsf{P}$, and note also that the NP-complete problem of determining whether $x \in 3\mathrm{SAT}$ is merely the question of whether $\#3\mathrm{CNF}(x) = 0$.

In apparent contrast to $\#\mathsf{P}$, all functions in $\#\mathsf{L}$ can be computed by NC circuits.

Theorem 7.1 *Relationships between counting classes.*

- $\mathsf{FP} \subseteq \#\mathsf{P} \subseteq \mathsf{FPSPACE}$
- $\mathsf{P}^{\mathsf{PP}} = \mathsf{P}^{\#\mathsf{P}}$ *(and thus also* $\mathsf{FP}^{\mathsf{NP}} \subseteq \mathsf{FP}^{\#\mathsf{P}}$*)*
- $\mathsf{FL} \subseteq \#\mathsf{L} \subseteq \mathsf{FNC}^2$.

It is not surprising that $\#\mathsf{P}$ and $\#\mathsf{L}$ capture the complexity of various functions that involve counting, but as the following examples illustrate, it sometimes is surprising which functions are difficult to compute.

The proof of Cook's Theorem that appears in Chapter 28 also proves that #3CNF is complete for $\#\mathsf{P}$, because it shows that for every nondeterministic polynomial-time machine M and every input x, one can efficiently construct a formula with the property that each accepting computation of M on input x corresponds to a distinct satisfying assignment, and vice versa. Thus the number of satisfying assignments equals the number of accepting computation paths. A reduction with this property is called *parsimonious.*

Most NP-complete languages that one encounters in practice are known to be complete under parsimonious reductions. (The reader may wish to check which of the reductions presented in Chapter 34 are parsimonious.) For any such complete language, it is clear how to define a corresponding complete function in $\#\mathsf{P}$.

Similarly, for the Graph Accessibility Problem (GAP), which is complete for NL, we can define the function that counts the number of paths from the start vertex s to the terminal vertex t. For reasons that will become clear soon, we consider two versions of this problem: one for general directed graphs, and one for directed acyclic graphs. (The restriction of GAP to acyclic graphs remains NL-complete.)

Number of Paths in a Graph (#Paths)

Instance: A directed graph on n vertices, with two vertices s and t.

Output: The number of simple paths from s to t. (A path is a *simple path* if it visits no vertex more than once.)

Number of Paths in a Directed Acyclic Graph (#DAG-Paths)

Instance: A directed acyclic graph on n vertices, with two vertices s and t.

Output: The number of paths from s to t. (In an acyclic graph, all paths are simple.)

As one might expect, the problem #DAG-Paths is complete for #L, but it may come as a surprise that #Paths is complete for #P, as shown by [Valiant, 1979]! That is, although it is easy to decide whether there is a path between two vertices, it seems quite difficult to count the number of distinct paths, unless the underlying graph is acyclic.

As another example of this phenomenon, consider the problem 2SAT, which is the same as 3SAT except that each clause has at most two literals. 2SAT is complete for NL, but the problem of counting the number of satisfying assignments for these formulas is complete for #P.

A striking illustration of the relationship between #P and #L is provided by the following two important problems from linear algebra.

Determinant

Instance: An integer matrix.

Output: The determinant of the matrix.

Recall that the determinant of a matrix M with entries $M_{i,j}$ is given by

$$\sum_{\pi} \operatorname{sign}(\pi) \prod_{i=1}^{n} M_{i,\pi(i)},$$

where the sum is over all permutations π on $\{1, \ldots, n\}$, and $\text{sign}(\pi)$ is -1 if π can be written as the composition of an odd number of transpositions, and $\text{sign}(\pi)$ is 1 otherwise.

PERMANENT

Instance: An integer matrix.

Output: The permanent of the matrix. The permanent of a matrix is given by

$$\sum_{\pi} \prod_{i=1}^{n} M_{i,\pi(i)}.$$

The reader is probably familiar with the determinant function, which can be computed efficiently by Gaussian elimination. The permanent may be less familiar, although its definition is formally simpler. Nobody has ever found an efficient way to compute the permanent, however.

We need to introduce slight modification of our function classes to classify these problems, however, because $\#\mathsf{L}$ and $\#\mathsf{P}$ consist of functions that take only non-negative values, whereas both the permanent and determinant can be negative.

Define GapL to be the class of functions that can be expressed as the difference of two $\#\mathsf{L}$ functions, and define GapP to be the difference of two $\#\mathsf{P}$ functions.

Theorem 7.2 *(a)* PERMANENT *is complete for* GapP*.*

(b) DETERMINANT *is complete for* GapL

The class of problems that are AC^0-Turing reducible to DETERMINANT is one of the most important subclasses of NC, and in fact contains most of the natural problems for which NC algorithms are known.

8 Kolmogorov Complexity

Until now, we have considered only dynamic complexity measures, namely, the time and space used by Turing machines. Kolmogorov complexity is a static complexity measure that captures the difficulty of describing a string. For example, the string consisting of three million zeroes can be described with fewer than three million symbols (as in this sentence). In contrast, for a string consisting of three million randomly generated bits, with high probability there is no shorter description than the string itself.

Let U be a universal Turing machine (see Chapter 26, Section 2.2). Let ϵ denote the empty string. The **Kolmogorov complexity** of a binary string y with respect to U, denoted by $K_U(y)$, is the length of the shortest binary string i such that on input $\langle i, \epsilon \rangle$, machine U outputs y. In essence, i is a description of y, for it tells U how to generate y.

The next theorem states that different choices for the universal Turing machine affect the definition of Kolmogorov complexity in only a small way.

Theorem 8.1 (Invariance Theorem) *There exists a universal Turing machine U such that for every universal Turing machine U', there is a constant c such that for all y, $K_U(y) \leq K_{U'}(y) + c$.*

Henceforth, let K be defined by the universal Turing machine of Theorem 8.1. For every integer n and every binary string y of length n, because y can be described by giving itself explicitly, $K(y) \leq n + c'$ for a constant c'. Call y **incompressible** if $K(y) \geq n$. Since there are 2^n binary strings of length n, and only $2^n - 1$ possible shorter descriptions, there exists an incompressible string for every length n.

Kolmogorov complexity gives a precise mathematical meaning to the intuitive notion of "randomness." If someone flips a coin fifty times and it comes up "heads" each time, then intuitively, the sequence of flips is not random—although from the standpoint of probability theory the all-heads sequence is precisely as likely as any other sequence. Probability theory does not provide the tools for calling one sequence "more random" than another; Kolmogorov complexity theory does.

Kolmogorov complexity provides a useful framework for presenting combinatorial arguments. For example, when one wants to prove that an object with some property P exists, then it is sufficient to show that any object that does *not* have property P has a short description; thus any incompressible (or "random") object must have property P. This sort of argument has been useful in proving lower bounds in complexity theory. For example, the paper [Dietzfelbinger *et al.*, 1991] uses Kolmogorov complexity to show that no Turing machine with a single worktape can compute the transpose of a matrix in less than time $\Omega(n^{3/2}/\sqrt{\log n})$.

9 Research Issues and Summary

As stated in the introduction to Chapter 27, the goals of complexity theory are (i) to ascertain the amount of computational resources required to solve important computational problems, and (ii) to classify problems according to their difficulty. The preceding two chapters have explained how complexity theory has devised a classification scheme in order to meet the second goal. The present chapter has presented a few of the additional notions of complexity that have been devised in order to capture more problems in this scheme. Progress toward the first goal (proving lower bounds) depends on knowing that levels in this classification scheme are in fact distinct. Thus the core research questions in complexity theory are expressed in terms of separating complexity classes:

- Is L different from NL?
- Is P different from RP or BPP?
- Is P different from NP?
- Is NP different from PSPACE?

Motivated by these questions, much current research is devoted to efforts to understand the power of nondeterminism, randomization, and interaction. In these studies, researchers have gone well beyond the theory presented in Chapters 27 through this chapter:

- beyond Turing machines and Boolean circuits, to restricted and specialized models in which nontrivial lower bounds on complexity can be proved;
- beyond deterministic reducibilities, to nondeterministic and probabilistic reducibilities, and refined versions of the reducibilities considered here;
- beyond worst case complexity, to average case complexity.

We have illustrated how recent research in complexity theory has had direct applications to other areas of computer science and mathematics. Probabilistically checkable proofs were used to show that obtaining approximate solutions to some optimization problems is as difficult as solving them exactly. Complexity theory provides new tools for studying questions in finite model theory, a branch of mathematical logic. Fundamental questions in complexity theory are intimately linked to practical questions about the use of cryptography for computer security, such as the existence of one-way functions and the strength of public key cryptosystems.

This last point illustrates the urgent practical need for progress in computational complexity theory. Many popular cryptographic systems in current use are based on unproven assumptions about the difficulty of computing certain functions (such as the factoring and discrete logarithm problems; see Chapters 38–42 of this *Handbook* for more background on cryptography). All of these systems are thus based on wishful thinking and conjecture. The need to resolve these open questions and replace conjecture with mathematical certainty should be self-evident. (In the brief history of complexity theory, we have learned that many popular conjectures turn out to be incorrect.)

With precisely defined models and mathematically rigorous proofs, research in complexity theory will continue to provide sound insights into the difficulty of solving real computational problems.

10 Defining Terms

Descriptive complexity: The study of classes of languages described by formulas in certain systems of logic.

Incompressible string: A string whose **Kolmogorov complexity** equals its length, so that it has no shorter encodings.

Interactive proof system: A protocol in which one or more *provers* try to convince another party called the *verifier* that the prover(s) possess certain true knowledge, such as the membership of a string x in a given language, often with the goal of revealing no further details about this knowledge. The prover(s) and verifier are formally defined as **probabilistic Turing machines** with special "interaction tapes" for exchanging messages.

Kolmogorov complexity: The minimum number of bits into which a string can be compressed without losing information. This is defined with respect to a fixed but *universal* decompression scheme, given by a universal Turing machine.

L-reduction A Karp reduction that preserves approximation properties of optimization problems.

Optimization problem: A computational problem in which the object is not to decide some yes/no property, as with a decision problem, but to find the best solution in those "yes" cases where a solution exists.

Polynomial hierarchy: The collection of classes of languages accepted by k-alternating Turing machines, over all $k \geq 0$ and with initial state existential or universal. The bottom level ($k = 0$) is the class P, and the next level ($k = 1$) comprises NP and $\mathsf{co\text{-}NP}$.

Polynomial time approximation scheme (PTAS): A meta-algorithm that for every $\epsilon > 0$ produces a polynomial time ϵ-approximation algorithm for a given optimization problem.

Probabilistic Turing machine: A Turing machine in which some transitions are random choices among finitely many alternatives.

Probabilistically checkable proof: An interactive proof system in which provers follow a fixed strategy, one not affected by any messages from the verifier. The prover's strategy for a given instance x of a decision problem can be represented by a finite oracle language B_x, which constitutes a proof of the correct answer for x.

Relational structure: The counterpart in formal logic of a data structure or class instance in the object-oriented sense. Examples are strings, directed graphs, and undirected graphs. Sets of relational structures generalize the notion of languages as sets of strings.

References

[Abiteboul and Vianu, 1991] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. *J. Comp. Sys. Sci.*, 43:62–124, 1991.

[Abiteboul and Vianu, 1995] S. Abiteboul and V. Vianu. Computing with first-order logic. *J. Comp. Sys. Sci.*, 50:309–335, 1995.

[Adleman, 1978] L. Adleman. Two theorems on random polynomial time. In *Proc. 19th Annual IEEE Symposium on Foundations of Computer Science*, pages 75–83, 1978.

[Allender, 1997] E. Allender. The permanent requires large uniform threshold circuits. Technical Report TR 97-51, DIMACS, September 1997. Submitted for publication.

[Alvarez and Jenner, 1993] C. Alvarez and B. Jenner. A very hard log-space counting class. *Theor. Comp. Sci.*, 107:3–30, 1993.

[Ambos-Spies, 1986] K. Ambos-Spies. A note on complete problems for complexity classes. *Inf. Proc. Lett.*, 23:227–230, 1986.

[Arora and Safra, 1992] S. Arora and S. Safra. Probabilistic checking of proofs. In *Proc. 33rd Annual IEEE Symposium on Foundations of Computer Science*, pages 2–13, 1992.

[Arora *et al.*, 1992] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and hardness of approximation problems. In *Proc. 33rd Annual IEEE Symposium on Foundations of Computer Science*, pages 14–23, 1992.

[Babai and Moran, 1988] L. Babai and S. Moran. Arthur-Merlin games: A randomized proof system, and a hierarchy of complexity classes. *J. Comp. Sys. Sci.*, 36:254–276, 1988.

[Babai *et al.*, 1991] L. Babai, L. Fortnow, and C. Lund. Nondeterministic exponential time has two-prover interactive protocols. *Computational Complexity*, 1:3–40, 1991. Addendum in vol. 2 of same journal.

[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.

[Bovet and Crescenzi, 1994] D. Bovet and P. Crescenzi. *Introduction to the Theory of Complexity.* Prentice Hall International (UK) Limited, Hertfordshire, U.K., 1994.

[Büchi, 1960] J. Büchi. Weak second-order arithmetic and finite automata. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 6:66–92, 1960.

[Chandra and Harel, 1982] A. Chandra and D. Harel. Structure and complexity of relational queries. *J. Comp. Sys. Sci.*, 25:99–128, 1982.

[Chandra *et al.*, 1981] A. Chandra, D. Kozen, and L. Stockmeyer. Alternation. *J. Assn. Comp. Mach.*, 28:114–133, 1981.

[Dietzfelbinger *et al.*, 1991] M. Dietzfelbinger, W. Maass, and G. Schnitger. The complexity of matrix transposition on one-tape off-line Turing machines. *Theor. Comp. Sci.*, 82:113–129, 1991.

[Downey and Fellows, 1995] R. Downey and M. Fellows. Fixed-parameter tractability and completeness I: Basic theory. *SIAM J. Comput.*, 24:873–921, 1995.

[Enderton, 1972] H.B. Enderton. *A mathematical introduction to logic.* Academic Press, New York, 1972.

[Fagin, 1974] R. Fagin. Generalized first-order spectra and polynomial-time recognizable sets. In R. Karp, editor, *Complexity of Computation: Proceedings of a Symposium in Applied Mathematics of the American Mathematical Society and the Society for Industrial and Applied Mathematics, Vol. VII*, pages 43–73. SIAM-AMS, 1974.

[Fagin, 1993] R. Fagin. Finite model theory—a personal perspective. *Theor. Comp. Sci.*, 116:3–31, 1993.

[Feige *et al.*, 1991] U. Feige, S. Goldwasser, L. Lovász, Safra S, and M. Szegedy. Approximating clique is almost NP-complete. In *Proc. 32nd Annual IEEE Symposium on Foundations of Computer Science*, pages 2–12, 1991.

[Feige *et al.*, 1996] U. Feige, S. Goldwasser, L. Lovász, Safra S, and M. Szegedy. Interactive proofs and the hardness of approximating cliques. *Journal of the ACM*, 43:268–292, 1996.

[Fortnow and Sipser, 1988] L. Fortnow and M. Sipser. Are there interactive protocols for co-NP languages? *Inf. Proc. Lett.*, 28:249–251, 1988.

[Gill, 1977] J. Gill. Computational complexity of probabilistic Turing machines. *SIAM J. Comput.*, 6:675–695, 1977.

[Goldwasser *et al.*, 1989] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. Comput.*, 18:186–208, 1989.

[Hartmanis, 1989] J. Hartmanis, editor. *Computational Complexity Theory.* American Mathematical Society, 1989.

[Hartmanis, 1994] J. Hartmanis. On computational complexity and the nature of computer science. *Comm. Assn. Comp. Mach.*, 37:37–43, 1994.

[Immerman, 1989] N. Immerman. Descriptive and computational complexity. In J. Hartmanis, editor, *Computational Complexity Theory*, volume 38 of *Proc. Symp. in Applied Math.*, pages 75–91. American Mathematical Society, 1989.

[Impagliazzo and Wigderson, 1997] R. Impagliazzo and A. Wigderson. P = BPP if E requires exponential circuits: Derandomizing the XOR Lemma. In *Proc. 29th Annual ACM Symposium on the Theory of Computing*, pages 220–229, 1997.

[Jones and Selman, 1974] N. Jones and A. Selman. Turing machines and the spectra of first-order formulas. *J. Assn. Comp. Mach.*, 39:139–150, 1974.

[Karp and Lipton, 1982] R. Karp and R. Lipton. Turing machines that take advice. *L'Enseignement Mathématique*, 28:191–210, 1982.

[Krentel, 1988] M. Krentel. The complexity of optimization problems. *J. Comp. Sys. Sci.*, 36:490–509, 1988.

[Lautemann, 1983] C. Lautemann. BPP and the polynomial hierarchy. *Inf. Proc. Lett.*, 17:215–217, 1983.

[Li and Vitányi, 1993] M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and its Applications*. Springer Verlag, 1993.

[Lindell, 1994] S. Lindell. How to define exponentiation from addition and multiplication in first-order logic on finite structures, 1994. Unpublished manuscript. The main result will appear in a forthcoming text by N. Immerman, *Descriptive and Computational Complexity*, in the Springer Verlag "Graduate Texts in Computer Science" series.

[Lund *et al.*, 1992] C. Lund, L. Fortnow, H. Karloff, and N. Nisan. Algebraic methods for interactive proof systems. *J. Assn. Comp. Mach.*, 39:859–868, 1992.

[Lutz, 1997] J. Lutz. The quantitative structure of exponential time. In L. Hemaspaandra and A. Selman, editors, *Complexity Theory Retrospective II*, pages 225–260. Springer Verlag, 1997.

[Lynch, 1982] J. Lynch. Complexity classes and theories of finite models. *Math. Sys. Thy.*, 15:127–144, 1982.

[McNaughton and Papert, 1971] R. McNaughton and S. Papert. *Counter-Free Automata.* MIT Press, Cambridge, MA, 1971.

[Papadimitriou and Yannakakis, 1984] C. Papadimitriou and M. Yannakakis. The complexity of facets (and some facets of complexity). *J. Comp. Sys. Sci.*, 28:244–259, 1984.

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

[Schützenberger, 1965] M.P. Schützenberger. On finite monoids having only trivial subgroups. *Inform. and Control*, 8:190–194, 1965.

[Shamir, 1992] A. Shamir. IP = PSPACE. *J. Assn. Comp. Mach.*, 39:869–877, 1992.

[Sipser, 1982] M. Sipser. On relativization and the existence of complete sets. In *Proc. 9th Annual International Conference on Automata, Languages, and Programming*, volume 140 of *Lect. Notes in Comp. Sci.*, pages 523–531. Springer Verlag, 1982.

[Sipser, 1983] M. Sipser. Borel sets and circuit complexity. In *Proc. 15th Annual ACM Symposium on the Theory of Computing*, pages 61–69, 1983.

[Sipser, 1992] M. Sipser. The history and status of the P versus NP question. In *Proc. 24th Annual ACM Symposium on the Theory of Computing*, pages 603–618, 1992.

[Stearns, 1990] R. Stearns. Juris Hartmanis: the beginnings of computational complexity. In A. Selman, editor, *Complexity Theory Retrospective*, pages 5–18. Springer Verlag, New York, 1990.

[Stockmeyer, 1976] L. Stockmeyer. The polynomial time hierarchy. *Theor. Comp. Sci.*, 3:1–22, 1976.

[Stockmeyer, 1987] L. Stockmeyer. Classifying the computational complexity of problems. *J. Symb. Logic*, 52:1–43, 1987.

[Toda, 1991] S. Toda. PP is as hard as the polynomial-time hierarchy. *SIAM J. Comput.*, 20:865–877, 1991.

[Valiant, 1979] L. Valiant. The complexity of computing the permanent. *Theor. Comp. Sci.*, 8:189–201, 1979.

[van Leeuwen, 1990] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science*, volume A. Elsevier and MIT Press, 1990.

[Vinay, 1991] V. Vinay. Counting auxiliary pushdown automata and semi-unbounded arithmetic circuits. In *Proc. 6th Annual IEEE Conference on Structure in Complexity Theory*, pages 270–284, 1991.

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

[Wang, 1997] J. Wang. Average-case computational complexity theory. In L. Hemaspaandra and A. Selman, editors, *Complexity Theory Retrospective II*, pages 295–328. Springer Verlag, 1997.

[Wrathall, 1976] C. Wrathall. Complete sets and the polynomial-time hierarchy. *Theor. Comp. Sci.*, 3:23–33, 1976.

Further Information

Primary sources for major theorems presented in this chapter include: Theorem 2.1 [Stockmeyer, 1976, Wrathall, 1976, Chandra *et al.*, 1981]; Theorem 3.2(a,b) [Gill, 1977], (c) [Sipser, 1983, Lautemann, 1983], (d) [Toda, 1991], (e) [Allender, 1997]; Theorem 3.3 [Adleman, 1978]; Theorem 4.1(a) [Abiteboul and Vianu, 1991], (b) [Stockmeyer, 1976], (c) [Fagin, 1974], (d,e,f) [Immerman, 1989], (g) ([Lindell, 1994], cf. [Barrington *et al.*, 1990]); Theorem 4.2(a) [Büchi, 1960], (b) [Schützenberger, 1965, McNaughton and Papert, 1971]; Theorem 5.1 [Shamir, 1992]; Theorem 5.2 [Babai *et al.*, 1991]; Theorem 5.3 [Arora and Safra, 1992]; Theorem 6.1 [Arora *et al.*, 1992]; Theorem 7.2(a) [Valiant, 1979], (b) [Vinay, 1991]. The operators in Definition 4.2 are from [Immerman, 1989] and [Abiteboul and Vianu, 1991]. Interactive proof systems were defined by [Goldwasser *et al.*, 1989], and in the "Arthur-Merlin" formulation, by [Babai and Moran, 1988]. A large compendium of optimization problems and hardness results collected by P. Crescenzi and V. Kann is available at:

`http://www.nada.kth.se/~viggo/index-en.html`

The class #P was introduced by [Valiant, 1979], and #L by [Alvarez and Jenner, 1993]. The book [Li and Vitányi, 1993] gives a far-reaching and comprehensive scholarly treatment of Kolmogorov complexity, with many applications, as well as the source of Theorem 8.1.

Three contemporary textbooks on complexity theory are [Balcázar *et al.*, 1990], [Bovet and Crescenzi, 1994], and [Papadimitriou, 1994]. [Wagner and Wechsung, 1986] is an exhaustive survey of complexity theory that covers work published before 1986. Another perspective of some of the issues covered in these three chapters may be found in the survey [Stockmeyer, 1987].

A good general reference is the *Handbook of Theoretical Computer Science* [van Leeuwen, 1990], volume A. The following chapters in the *Handbook* are particularly relevant: "Machine models and simulations," by P. van Emde Boas, pp. 1–66; "A catalog of complexity classes," by D. S. Johnson, pp. 67–161; "Machine-independent complexity theory," by J. I. Seiferas, pp. 163–186; "Kolmogorov complexity and its applications," by M. Li and P. M. B. Vitányi, pp. 187–254; and "The complexity of finite functions," by R. B. Boppana and M. Sipser, pp. 757–804, which covers circuit complexity.

A collection of articles edited by Hartmanis [Hartmanis, 1989] includes an overview of complexity theory, and chapters on sparse complete languages, on relativizations, on interactive proof systems, and on applications of complexity theory to cryptography. For historical perspectives on complexity theory, see [Hartmanis, 1994], [Sipser, 1992], and [Stearns, 1990].

There are many areas of complexity theory that we have not been able to cover in these chapters. Some of them cross-pollinate with other fields of computer science and are reflected in other chapters of this *Handbook*. Three others are *average-case complexity*, *resource-bounded measure theory*, and *parameterized complexity*. Recent surveys on the first two are [Wang, 1997, Lutz, 1997], while the third stems from [Downey and Fellows, 1995] and has its own Web site, currently maintained by M. Hallett at:

`http://csr.uvic.ca/home/mhallett/W.hier/compendium.html`

Surveys and lecture notes on complexity theory that can be obtained via WWW are maintained by A. Czumaj and M. Kutylowski at:

`http://www.uni-paderborn.de/fachbereich/AG/agmadh/WWW/english/scripts.html`

As usual with the WWW, these links are subject to change. A good stem page to begin searches is the site for SIGACT, the ACM Special Interest Group on Algorithms and Computation Theory:

`http://sigact.acm.org/`

This has a pointer to a "Virtual Address Book" that indexes the personal Web pages of over 1,000 computer scientists, including all three authors of these chapters. Many of these pages have downloadable papers and links to further research resources.

Research papers on complexity theory are presented at several annual conferences, including the annual ACM Symposium on Theory of Computing; the annual International Colloquium on Automata, Languages, and Programming, sponsored by the European Association for Theoretical Computer Science (EATCS); and the annual Symposium on Foundations of Computer Science, sponsored by the IEEE. The annual Conference on Computational Complexity (formerly Structure in Complexity Theory), also sponsored by the IEEE, is entirely devoted to complexity theory. Research articles on complexity theory regularly appear in the following journals, among others: *Chicago Journal on Theoretical Computer Science, Computational Complexity, Information and Computation, Journal of the ACM, Journal of Computer and System Sciences, Mathematical Systems Theory, SIAM Journal on Computing,* and *Theoretical Computer Science.* Each issue of *ACM SIGACT News* and *Bulletin of the EATCS* contains a column on complexity theory.