Complexity Theory

Eric W. 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

January 25, 2012

1 Introduction

Computational complexity is the study of the difficulty of solving computational problems, in terms of the required computational resources, such as time and space (memory). Whereas the analysis of algorithms focuses on the time or space requirements of an *individual* algorithm for a *specific* problem (such as sorting), complexity theory focuses on the **complexity class** of problems solvable in the same amount of time or space. Most common computational problems fall into a small number of complexity classes. Two important complexity classes are P, the set of problems that can be solved in polynomial time, and NP, the set of problems whose solutions can be verified in polynomial time.

By quantifying the resources required to solve a problem, complexity theory has profoundly affected our thinking about computation. Computability theory establishes the existence of undecidable problems, which cannot be solved in principle regardless of the amount of time invested. However, computability theory fails to find meaningful distinctions among decidable problems. In contrast, complexity theory establishes the existence of decidable problems that, although solvable in principle, cannot be solved in practice, because the time and space required would be larger than the age and size of the known universe [Stockmeyer and Chandra, 1979]. Thus, complexity theory characterizes the computationally feasible problems.

The quest for the boundaries of the set of feasible problems has led to the most important unsolved question in all of computer science: Is P different from NP? Hundreds of fundamental problems, including many ubiquitous optimization problems of operations research, are **NP-complete**; they are the hardest problems in NP. If someone could find a polynomial-time algorithm for any one NP-complete problem, then there would be polynomial-time algorithms for all of them. Despite the concerted efforts of many scientists over several decades, no polynomial-time algorithm has been found for any NP-complete problem. Although we do not yet know whether P is different from NP, showing that a problem is NP-complete provides strong evidence that the problem is computationally infeasible and justifies the use of heuristics for solving the problem.

In this chapter, we define P, NP, and related complexity classes. We illustrate the use of **diagonalization** and **padding** techniques to prove relationships between classes. Next, we define NP-completeness, and we show how to prove that a problem is NP-complete. Finally, we define complexity classes for probabilistic and interactive computations.

Throughout this chapter, all numeric functions take integer arguments and produce integer values. All logarithms are taken to base 2. In particular, $\log n$ means $\lceil \log_2 n \rceil$.

[1] Supported by the National Science Foundation under Grants CCF-0832787 and CCF-1064785.

[2] Supported by the National Science Foundation under Grants IIS-0832843, CNS-0851957, and DUE-1044207.

[3] Supported by the National Science Foundation under Grant CCR-9821040.

2 Models of Computation

To develop a theory of the difficulty of computational problems, we need to specify precisely what a problem is, what an algorithm is, and what a measure of difficulty is. For simplicity, complexity theorists have chosen to represent problems as languages (i.e., as sets of strings of symbols), to model algorithms by off-line multitape **Turing machines**, and to measure computational difficulty by the time and space required by a Turing machine. To justify these choices, some theorems of complexity theory show how to translate statements about, say, the time complexity of language recognition by Turing machines into statements about computational problems on more realistic models of computation. These theorems imply that the principles of complexity theory are not artifacts of Turing machines, but instead are intrinsic properties of computation.

This section defines different kinds of Turing machines. The *deterministic* Turing machine models actual computers. The *nondeterministic* Turing machine is not a realistic model, but it helps classify the complexity of important computational problems. The *alternating* Turing machine models a form of parallel computation, and it helps elucidate the relationship between time and space.

2.1 Computational Problems and Languages

Computer scientists have invented many elegant formalisms for representing data and control structures. Fundamentally, all representations are patterns of symbols. Therefore, we represent an instance of a computational problem as a sequence of symbols.

Let Σ be a finite set, called the *alphabet*. A *word* over Σ is a finite sequence of symbols from Σ. Sometimes a word is called a *string*. Let Σ^* denote the set of all words over Σ. For example, if $\Sigma = \{\mathbf{0},\mathbf{1}\}$, then

$$\Sigma^* = \{\lambda, \mathbf{0}, \mathbf{1}, \mathbf{00}, \mathbf{01}, \mathbf{10}, \mathbf{11}, \mathbf{000}, \ldots\}$$

is the set of all binary words, including the empty word λ. The *length* of a word w, denoted by $|w|$, is the number of symbols in w. A *language* over Σ is a subset of Σ^*.

A *decision problem* is a computational problem whose answer is simply `yes` or `no`. For example: Is the input graph connected? or Is the input a sorted list of integers? A decision problem can be expressed as a membership problem for a language A: for an input x, does x belong to A? For a language A that represents connected graphs, the input word x might represent an input graph G, and $x \in A$ if and only if G is connected.

For every decision problem, the representation should allow for easy parsing, to determine whether a word represents a legitimate instance of the problem. Furthermore, the representation should be concise. In particular, it would be unfair to encode the answer to the problem into the representation of an instance of the problem; for example, for the problem of deciding whether an input graph is connected, the representation should not have an extra bit that tells whether the graph is connected. A set of integers $S = \{x_1, \ldots, x_m\}$ is represented by listing the binary representation of each x_i, with the representations of consecutive integers in S separated by a nonbinary symbol. A graph is naturally represented by giving either its adjacency matrix or a set of adjacency lists, where the list for each vertex v specifies the vertices adjacent to v.

Whereas the solution to a decision problem is `yes` or `no`, the solution to an optimization problem is more complicated; for example, determine the shortest path from vertex u to vertex v in an input graph G. Nevertheless, for every optimization (minimization) problem, with objective function g, there is a corresponding decision problem that asks whether there exists a feasible solution z such that $g(z) \leq k$, where k is a given target value. Clearly, if there is an algorithm that solves an optimization problem, then that algorithm can be used to solve the corresponding decision problem. Conversely, if algorithm solves the decision problem, then with a binary search on the range of values of g, we can determine the optimal value. Moreover, using a decision problem as a subroutine often enables us to construct an optimal solution; for

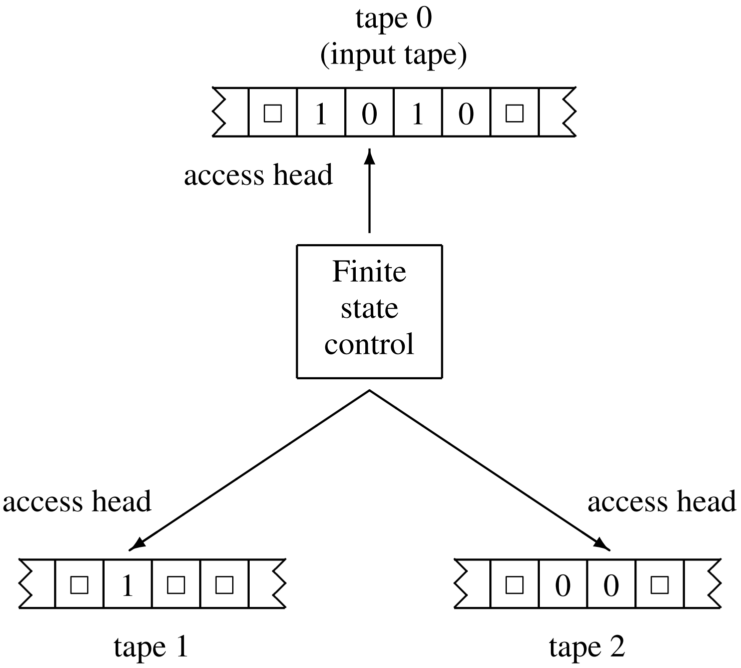

Figure 1: A 2-tape Turing machine.

.

example, if we are trying to find a shortest path, we can use a decision problem that determines if a shortest path starting from a given vertex uses a given edge. Therefore, there is little loss of generality in considering only decision problems, represented as language membership problems.

2.2 Turing Machines

This subsection and the next three give precise, formal definitions of Turing machines and their variants. These subsections are intended for reference. For the rest of this chapter, the reader need not understand these definitions in detail, but may generally substitute “program” or “computer” for each reference to “Turing machine”.

A k-worktape **Turing machine** M consists of the following:

- A finite set of states Q, with special states q_0 (initial state), q_A (accept state), and q_R (reject state).
- A finite alphabet Σ, and a special blank symbol $\square \notin \Sigma$.
- The $k+1$ linear tapes, each divided into cells. Tape 0 is the *input tape*, and tapes $1, \ldots, k$ are the *worktapes*. Each tape is infinite to the left and to the right. Each cell holds a single symbol from $\Sigma \cup \{\square\}$. By convention, the input tape is read only. Each tape has an access head, and at every instant, each access head scans one cell. See Fig. 1.
- A finite transition table δ, which comprises tuples of the form

$$(q, s_0, s_1, \ldots, s_k, q', s'_1, \ldots, s'_k, d_0, d_1, \ldots, d_k)$$

where $q, q' \in Q$, each $s_i, s'_i \in \Sigma \cup \{\Box\}$, and each $d_i \in \{-1, 0, +1\}$.

A tuple specifies a step of M: if the current state is q, and $s_0, s_1, \ldots, s_k$ are the symbols in the cells scanned by the access heads, then M replaces s_i by s'_i for $i = 1, \ldots, k$ simultaneously, changes state to q', and moves the head on tape i one cell to the left ($d_i = -1$) or right ($d_i = +1$) or not at all ($d_i = 0$) for $i = 0, \ldots, k$. Note that M cannot write on tape 0, that is, M may write only on the worktapes, not on the input tape.

- No tuple contains q_A or q_R as its first component. Thus, once M enters state q_A or state q_R, it stops.
- Initially, M is in state q_0, an input word in Σ^* is inscribed on contiguous cells of the input tape, the access head on the input tape is on the leftmost symbol of the input word, and all other cells of all tapes contain the blank symbol $\Box$.

The Turing machine M that we have defined is *nondeterministic:* δ may have several tuples with the same combination of state q and symbols $s_0, s_1, \ldots, s_k$ as the first $k+2$ components, so that M may have several possible next steps. A machine M is *deterministic* if for every combination of state q and symbols $s_0, s_1, \ldots, s_k$, at most one tuple in δ contains the combination as its first $k+2$ components. A deterministic machine always has at most one possible next step.

A *configuration* of a Turing machine M specifies the current state, the contents of all tapes, and the positions of all access heads.

A *computation path* is a sequence of configurations $C_0, C_1, \ldots, C_t, \ldots$, where C_0 is the initial configuration of M, and each C_{j+1} follows from C_j in one step by applying the changes specified by a tuple in δ. If no tuple is applicable to C_t, then C_t is *terminal*, and the computation path is *halting.* If M has no infinite computation paths, then M *always halts.*

A halting computation path is *accepting* if the state in the last configuration C_t is q_A; otherwise it is *rejecting*. By adding tuples to the program if needed, we can ensure that every rejecting computation ends in state q_R. This leaves the question of computation paths that do not halt. In complexity theory we rule this out by considering only machines whose computation paths *always halt*. M *accepts* an input word x if there exists an accepting computation path that starts from the initial configuration in which x is on the input tape. For nondeterministic M, it does not matter if some other computation paths end at q_R. If M is deterministic, then there is at most one halting computation path, hence at most one accepting path.

The *language accepted by* M, written $L(M)$, is the set of words accepted by M. If $A = L(M)$, and M always halts, then M *decides* A.

In addition to deciding languages, deterministic Turing machines can compute functions. Designate tape 1 to be the *output tape*. If M halts on input word x, then the nonblank word on tape 1 in the final configuration is the output of M. A function f is *computable* if there exists a deterministic Turing machine M that always halts such that for each input word x, the output of M is the value of $f(x)$.

Almost all results in complexity theory are insensitive to minor variations in the underlying computational models. For example, we could have chosen Turing machines whose tapes are restricted to be only one-way infinite or whose alphabet is restricted to $\{\mathbf{0},\mathbf{1}\}$. It is straightforward to simulate a Turing machine as defined by one of these restricted Turing machines, one step at a time: each step of the original machine can be simulated by $O(1)$ steps of the restricted machine.

2.3 Universal Turing Machines

Chapter *** CHAPTER REFERENCE: FORMAL MODELS AND COMPUTABILITY *** stated that there exists a *universal Turing machine* U, which takes as input a string $\langle M, x\rangle$ that encodes a Turing machine M and a word x, and simulates the operation of M on x, and U accepts $\langle M, x\rangle$ if and only if M

accepts x. For our purposes here, we also need that U simulates M *efficiently*: A theorem of Hennie and Stearns [1966] implies that the machine U can be constructed to have only two worktapes, such that U can simulate any t steps of M in only $O(t \log t)$ steps of its own, using only $O(1)$ times the worktape cells used by M. The constants implicit in these big-O bounds may depend on M.

We can think of U with a fixed M as a machine U_M and define $L(U_M) = \{x : U \text{ accepts } \langle M, x\rangle\}$. Then $L(U_M) = L(M)$. If M always halts, then U_M always halts, and if M is deterministic, then U_M is deterministic.

2.4 Alternating Turing Machines

By definition, a nondeterministic Turing machine M accepts its input word x if there exists an accepting computation path, starting from the initial configuration with x on the input tape. Let us call a *configuration* C accepting if there is a computation path of M that starts in C and ends in a configuration whose state is q_A. Equivalently, a configuration C is accepting if either the state in C is q_A or there exists an accepting configuration C' reachable from C by one step of M. Then M accepts x if the initial configuration with input word x is accepting.

The *alternating Turing machine* generalizes this notion of acceptance. In an alternating Turing machine M, each state is labeled either existential or universal. (Do not confuse the universal state in an alternating Turing machine with the universal Turing machine.) A nonterminal configuration C is existential (respectively, universal) if the state in C is labeled existential (universal). A terminal configuration is accepting if its state is q_A. A nonterminal existential configuration C is accepting if there *exists* an accepting configuration C' reachable from C by one step of M. A nonterminal universal configuration C is accepting if for *every* configuration C' reachable from C by one step of M, the configuration C' is accepting. Finally, M accepts x if the initial configuration with input word x is an accepting configuration.

A nondeterministic Turing machine is thus a special case of an alternating Turing machine in which every state is existential.

The computation of an alternating Turing machine M alternates between existential states and universal states. Intuitively, from an existential configuration, M guesses a step that leads toward acceptance; from a universal configuration, M checks whether each possible next step leads toward acceptance—in a sense, M checks all possible choices in parallel. An alternating computation captures the essence of a two-player game: player 1 has a winning strategy if there exists a move for player 1 such that for every move by player 2, there exists a subsequent move by player 1, etc., such that player 1 eventually wins.

2.5 Oracle Turing Machines

Complexity theoreticians have found it very useful to have a mechanism to describe when it is helpful to have a subroutine for some problem A, when trying to solve another problem. The language A is called an **oracle**. Conceptually, an algorithm queries the oracle whether a word w is in A, and it receives the correct answer in one step.

An *oracle Turing machine* is a Turing machine M with a special *oracle tape* and special states QUERY, YES, and NO. The computation of the oracle Turing machine M^A, with oracle language A, is the same as that of an ordinary Turing machine, except that when M enters the QUERY state with a word w on the oracle tape, in one step, M enters either the YES state if $w \in A$ or the NO state if $w \notin A$. Furthermore, during this step, the oracle tape is erased, so that the time for setting up each query is accounted separately.

3 Resources and Complexity Classes

In this section, we define the measures of difficulty of solving computational problems. We introduce complexity classes, which enable us to classify problems according to the difficulty of their solution.

3.1 Time and Space

We measure the difficulty of a computational problem by the running time and the space (memory) requirements of an algorithm that solves the problem.

We express the complexity of a problem, in terms of the growth of the required time or space, as a function of the length n of the input word that encodes a problem instance. We consider the worst-case complexity, that is, for each n, the maximum time or space required among all inputs of length n.

Let M be a Turing machine that always halts. The *time* taken by M on input word x, denoted by $\text{Time}_M(x)$, is defined as follows:

- If M accepts x, then $\text{Time}_M(x)$ is the number of steps in the shortest accepting computation path for x.
- If M rejects x, then $\text{Time}_M(x)$ is the number of steps in the longest computation path for x.

For a deterministic machine M, for every input x, there is at most one halting computation path, and its length is $\text{Time}_M(x)$. For a nondeterministic machine M, if $x \in L(M)$, then M can guess the correct steps to take toward an accepting configuration, and $\text{Time}_M(x)$ measures the length of the path on which M always makes the best guess.

The *space* used by a Turing machine M on input x, denoted $\text{Space}_M(x)$, is defined as follows. The space used by a halting computation path is the number of worktape cells visited by the worktape heads of M during the computation path. Because the space occupied by the input word is not counted, a machine can use a sublinear ($o(n)$) amount of space.

- If M accepts x, then $\text{Space}_M(x)$ is the minimum space used among all accepting computation paths for x.
- If M rejects x, then $\text{Space}_M(x)$ is the maximum space used among all computation paths for x.

The **time complexity** of a machine M is the function

$$t(n) = \max\{\text{Time}_M(x) : |x| = n\}$$

We assume that M reads all of its input word, and the blank symbol after the right end of the input word, so $t(n) \geq n + 1$. The **space complexity** of M is the function

$$s(n) = \max\{\text{Space}_M(x) : |x| = n\}$$

Because few interesting languages can be decided by machines of sublogarithmic space complexity, we henceforth assume that $s(n) \geq \log n$.

A function $f(x)$ is *computable in polynomial time* if there exists a deterministic Turing machine M of polynomial time complexity such that for each input word x, the output of M is $f(x)$.

3.2 Complexity Classes

Having defined the time complexity and space complexity of individual Turing machines, we now define classes of languages with particular complexity bounds. These definitions will lead to definitions of P and NP.

Let $t(n)$ and $s(n)$ be numeric functions. Define the following classes of languages.

- $\mathsf{DTIME}[t(n)]$ is the class of languages decided by deterministic Turing machines of time complexity $O(t(n))$.
- $\mathsf{NTIME}[t(n)]$ is the class of languages decided by nondeterministic Turing machines of time complexity $O(t(n))$.
- $\mathsf{DSPACE}[s(n)]$ is the class of languages decided by deterministic Turing machines of space complexity $O(s(n))$.
- $\mathsf{NSPACE}[s(n)]$ is the class of languages decided by nondeterministic Turing machines of space complexity $O(s(n))$.

We sometimes abbreviate $\mathsf{DTIME}[t(n)]$ to $\mathsf{DTIME}[t]$ (and so on) when t is understood to be a function, and when no reference is made to the input length n.

The following are the **canonical complexity classes**:

- $\mathsf{L} = \mathsf{DSPACE}[\log n]$ (deterministic log space)
- $\mathsf{NL} = \mathsf{NSPACE}[\log n]$ (nondeterministic log space)
- $\mathsf{P} = \mathsf{DTIME}[n^{O(1)}] = \bigcup_{k\geq 1} \mathsf{DTIME}[n^k]$ (polynomial time)
- $\mathsf{NP} = \mathsf{NTIME}[n^{O(1)}] = \bigcup_{k\geq 1} \mathsf{NTIME}[n^k]$ (nondeterministic polynomial time)
- $\mathsf{PSPACE} = \mathsf{DSPACE}[n^{O(1)}] = \bigcup_{k\geq 1} \mathsf{DSPACE}[n^k]$ (polynomial space)
- $\mathsf{E} = \mathsf{DTIME}[2^{O(n)}] = \bigcup_{k\geq 1} \mathsf{DTIME}[k^n]$
- $\mathsf{NE} = \mathsf{NTIME}[2^{O(n)}] = \bigcup_{k\geq 1} \mathsf{NTIME}[k^n]$
- $\mathsf{EXP} = \mathsf{DTIME}[2^{n^{O(1)}}] = \bigcup_{k\geq 1} \mathsf{DTIME}[2^{n^k}]$ (deterministic exponential time)
- $\mathsf{NEXP} = \mathsf{NTIME}[2^{n^{O(1)}}] = \bigcup_{k\geq 1} \mathsf{NTIME}[2^{n^k}]$ (nondeterministic exponential time)

The class PSPACE is defined in terms of the DSPACE complexity measure. By Savitch's Theorem (see Theorem 4.2), the NSPACE measure with polynomial bounds also yields PSPACE.

The class P contains many familiar problems that can be solved efficiently, such as (decision problem versions of) finding shortest paths in networks, parsing for context-free languages, sorting, matrix multiplication, and linear programming. Consequently, P has become accepted as representing the set of computationally feasible problems. Although one could legitimately argue that a problem whose best algorithm has time complexity $\Theta(n^{99})$ is really infeasible, in practice, the time complexities of the vast majority of natural polynomial-time problems have low degrees: they have algorithms that run in $O(n^4)$ time or less. Moreover, P is a robust class: though defined by Turing machines, P remains the same when defined by other models of sequential computation. For example, random access machines (RAMs) (a more realistic model of computation defined in Chapter *** CHAPTER REFERENCE: FORMAL MODELS AND COMPUTABILITY ***) can be used to define P, because Turing machines and RAMs can simulate each other with polynomial-time overhead.

The class NP can also be defined by means other than nondeterministic Turing machines. NP equals the class of problems whose solutions can be *verified* quickly, by deterministic machines in polynomial time. Equivalently, NP comprises those languages whose membership proofs can be checked quickly.

For example, one language in NP is the set of satisfiable Boolean formulas, called SAT. A Boolean formula ϕ is satisfiable if there exists a way of assigning `true` or `false` to each variable such that under this truth assignment, the value of ϕ is `true`. For example, the formula $x \wedge (\overline{x} \vee y)$ is satisfiable, but $x \wedge \overline{y} \wedge (\overline{x} \vee y)$ is not satisfiable. A nondeterministic Turing machine M, after checking the syntax of ϕ and counting the number n of variables, can nondeterministically write down an n-bit 0-1 string a on its tape, and then deterministically (and easily) evaluate ϕ for the truth assignment denoted by a. The computation path corresponding to each individual a accepts if and only if $\phi(a) = \texttt{true}$, and so M itself accepts ϕ if and only if ϕ is satisfiable; i.e., $L(M) = \text{SAT}$. Again, this checking of given assignments differs significantly from trying to *find* an accepting assignment.

Another language in NP is the set of undirected graphs with a *Hamiltonian circuit*, i.e., a path of edges that visits each vertex exactly once and returns to the starting point. If a solution exists and is given, its correctness can be verified quickly. Finding such a circuit, however, or proving one does not exist, appears to be computationally difficult.

The characterization of NP as the set of problems with easily verified solutions is formalized as follows: $A \in \mathsf{NP}$ if and only if there exist a language $A' \in \mathsf{P}$ and a polynomial p such that for every x, $x \in A$ if and only if there exists a y such that $|y| \leq p(|x|)$ and $(x, y) \in A'$. Here, whenever x belongs to A, y is interpreted as a positive solution to the problem represented by x, or equivalently, as a proof that x belongs to A. The difference between P and NP is that between solving and checking, or between finding a proof of a mathematical theorem and testing whether a candidate proof is correct. In essence, NP represents all sets of theorems with proofs that are short (i.e., of polynomial length) and checkable quickly (i.e., in polynomial time), while P represents those statements that can proved or refuted quickly from scratch.

Further motivation for studying L, NL, and PSPACE, comes from their relationships to P and NP. Namely, L and NL are the largest space-bounded classes known to be contained in P, and PSPACE is the smallest space-bounded class known to contain NP. (It is worth mentioning here that NP does not stand for "non-polynomial time"; the class P is a subclass of NP.) Similarly, EXP is of interest primarily because it is the smallest deterministic time class known to contain NP. The closely-related class E is not known to contain NP.

4 Relationships Between Complexity Classes

The P versus NP question asks about the relationship between these complexity classes: Is P a proper subset of NP, or does $\mathsf{P} = \mathsf{NP}$? Much of complexity theory focuses on the relationships between complexity classes, because these relationships have implications for the difficulty of solving computational problems. In this section, we summarize important known relationships. We demonstrate two techniques for proving relationships between classes: diagonalization and padding.

4.1 Constructibility

The most basic theorem that one should expect from complexity theory would say, "If you have more resources, you can do more." Unfortunately, if we aren't careful with our definitions, then this claim is false:

Theorem 4.1 (Gap Theorem) *There is a computable, strictly increasing time bound $t(n)$ such that* $\mathsf{DTIME}[t(n)] = \mathsf{DTIME}[2^{2^{t(n)}}]$ [Borodin, 1972].

That is, there is an empty gap between time $t(n)$ and time doubly-exponentially greater than $t(n)$, in the sense that anything that can be computed in the larger time bound can already be computed in the smaller time bound. That is, even with much more time, you can't compute more. This gap can be made much larger than doubly-exponential; for any computable r, there is a computable time bound t such that $\mathsf{DTIME}[t(n)] = \mathsf{DTIME}[r(t(n))]$. Exactly analogous statements hold for the NTIME, DSPACE, and NSPACE measures.

Fortunately, the gap phenomenon cannot happen for time bounds t that anyone would ever be interested in. Indeed, the proof of the Gap Theorem proceeds by showing that one can define a time bound t such that no machine has a running time that is between $t(n)$ and $2^{2^{t(n)}}$. This theorem indicates the need for formulating only those time bounds that actually describe the complexity of some machine.

A function $t(n)$ is **time-constructible** if there exists a deterministic Turing machine that halts after exactly $t(n)$ steps for every input of length n. A function $s(n)$ is **space-constructible** if there exists a deterministic Turing machine that uses exactly $s(n)$ worktape cells for every input of length n. (Most authors consider only functions $t(n) \geq n + 1$ to be time-constructible, and many limit attention to $s(n) \geq \log n$ for space bounds. There do exist sub-logarithmic space-constructible functions, but we prefer to avoid the tricky theory of $o(\log n)$ space bounds.)

For example, $t(n) = n+1$ is time-constructible. Furthermore, if $t_1(n)$ and $t_2(n)$ are time-constructible, then so are the functions $t_1 + t_2$, $t_1 t_2$, $t_1^{t_2}$, and c^{t_1} for every integer $c > 1$. Consequently, if $p(n)$ is a polynomial, then $p(n) = \Theta(t(n))$ for some time-constructible polynomial function $t(n)$. Similarly, $s(n) = \log n$ is space-constructible, and if $s_1(n)$ and $s_2(n)$ are space-constructible, then so are the functions $s_1 + s_2$, $s_1 s_2$, $s_1^{s_2}$, and c^{s_1} for every integer $c > 1$. Many common functions are space-constructible: e.g., $n \log n$, n^3, 2^n, $n!$.

Constructibility helps eliminate an arbitrary choice in the definition of the basic time and space classes. For general time functions t, the classes $\mathsf{DTIME}[t]$ and $\mathsf{NTIME}[t]$ may vary depending on whether machines are required to halt within t steps on all computation paths, or just on those paths that accept. If t is time-constructible and s is space-constructible, however, then $\mathsf{DTIME}[t]$, $\mathsf{NTIME}[t]$, $\mathsf{DSPACE}[s]$, and $\mathsf{NSPACE}[s]$ can be defined without loss of generality in terms of Turing machines that always halt.

As a general rule, any function $t(n) \geq n + 1$ and any function $s(n) \geq \log n$ that one is interested in as a time or space bound, is time- or space-constructible, respectively. As we have seen, little of interest can be proved without restricting attention to constructible functions. This restriction still leaves a rich class of resource bounds.

4.2 Basic Relationships

Clearly, for all time functions $t(n)$ and space functions $s(n)$, $\mathsf{DTIME}[t(n)] \subseteq \mathsf{NTIME}[t(n)]$ and $\mathsf{DSPACE}[s(n)] \subseteq \mathsf{NSPACE}[s(n)]$, because a deterministic machine is a special case of a nondeterministic machine. Furthermore, $\mathsf{DTIME}[t(n)] \subseteq \mathsf{DSPACE}[t(n)]$ and $\mathsf{NTIME}[t(n)] \subseteq \mathsf{NSPACE}[t(n)]$, because at each step, a k-tape Turing machine can write on at most $k = O(1)$ previously unwritten cells. The next theorem presents additional important relationships between classes.

Theorem 4.2 *Let $t(n)$ be a time-constructible function, and let $s(n)$ be a space-constructible function, $s(n) \geq \log n$.*

(a) $\mathsf{NTIME}[t(n)] \subseteq \mathsf{DTIME}[2^{O(t(n))}]$.

(b) $\mathsf{NSPACE}[s(n)] \subseteq \mathsf{DTIME}[2^{O(s(n))}]$.

(c) $\mathsf{NTIME}[t(n)] \subseteq \mathsf{DSPACE}[t(n)]$.

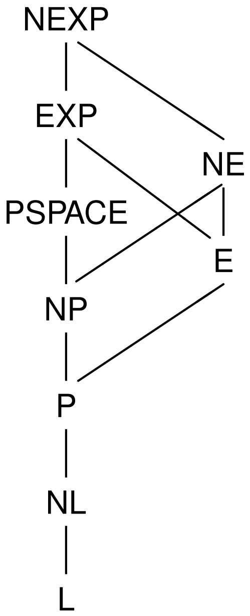

Figure 2: Inclusion relationships between the canonical complexity classes.

(d) **(Savitch's Theorem)** $\mathsf{NSPACE}[s(n)] \subseteq \mathsf{DSPACE}[s(n)^2]$ [Savitch, 1970].

As a consequence of the first part of this theorem, $\mathsf{NP} \subseteq \mathsf{EXP}$. No better general upper bound on deterministic time is known for languages in NP. See Figure 2 for other known inclusion relationships between canonical complexity classes.

Although we do not know whether allowing nondeterminism strictly increases the class of languages decided in polynomial time, Savitch's Theorem says that for space classes, nondeterminism does not help by more than a polynomial amount.

4.3 Complementation

For a language A over an alphabet Σ, define $\overline{A}$ to be the complement of A in the set of words over Σ: i.e., $\overline{A} = \Sigma^* - A$. For a class of languages $\mathcal{C}$, define co-$\mathcal{C}$ = $\{\, \overline{A} : A \in \mathcal{C} \,\}$. If $\mathcal{C}$ = co-$\mathcal{C}$, then $\mathcal{C}$ is **closed under complementation.**

In particular, co-NP is the class of languages that are complements of languages in NP. For the language SAT of satisfiable Boolean formulas, $\overline{\text{SAT}}$ is essentially the set of unsatisfiable formulas, whose value is `false` for every truth assignment, together with the syntactically incorrect formulas. A closely related language in co-NP is the set of Boolean tautologies, namely, those formulas whose value is `true` for every truth assignment. The question of whether NP equals co-NP comes down to whether every tautology has a short (i.e., polynomial-sized) proof. The only obvious general way to prove a tautology ϕ in m variables is to verify all 2^m rows of the truth table for ϕ, taking exponential time. Most complexity theorists believe that there is no general way to reduce this time to polynomial, hence that $\mathsf{NP} \neq$ co-NP.

Questions about complementation bear directly on the P vs. NP question. It is easy to show that P is closed under complementation (see the next theorem). Consequently, if $\mathsf{NP} \neq \text{co-}\mathsf{NP}$, then $\mathsf{P} \neq \mathsf{NP}$.

Theorem 4.3 (Complementation Theorems) *Let t be a time-constructible function, and let s be a space-constructible function, with $s(n) \geq \log n$ for all n. Then*

1. $\mathsf{DTIME}[t]$ *is closed under complementation.*

2. $\mathsf{DSPACE}[s]$ *is closed under complementation.*

3. $\mathsf{NSPACE}[s]$ *is closed under complementation* [Immerman, 1988; Szelepcsényi, 1988].

The Complementation Theorems are used to prove the Hierarchy Theorems in the next section.

4.4 Hierarchy Theorems and Diagonalization

A hierarchy theorem is a theorem that says "If you have more resources, you can compute more." As we saw in Section 4.1, this theorem is possible only if we restrict attention to constructible time and space bounds. Next, we state hierarchy theorems for deterministic and nondeterministic time and space classes. In the following, $\subset$ denotes *strict* inclusion between complexity classes.

Theorem 4.4 (Hierarchy Theorems) *Let t_1 and t_2 be time-constructible functions, and let s_1 and s_2 be space-constructible functions, with $s_1(n), s_2(n) \geq \log n$ for all n.*

(a) If $t_1(n) \log t_1(n) = o(t_2(n))$, then $\mathsf{DTIME}[t_1] \subset \mathsf{DTIME}[t_2]$.

(b) If $t_1(n+1) = o(t_2(n))$, then $\mathsf{NTIME}[t_1] \subset \mathsf{NTIME}[t_2]$ [Seiferas et al., 1978].

(c) If $s_1(n) = o(s_2(n))$, then $\mathsf{DSPACE}[s_1] \subset \mathsf{DSPACE}[s_2]$.

(d) If $s_1(n) = o(s_2(n))$, then $\mathsf{NSPACE}[s_1] \subset \mathsf{NSPACE}[s_2]$.

As a corollary of the Hierarchy Theorem for DTIME,

$$\mathsf{P} \subseteq \mathsf{DTIME}[n^{\log n}] \subset \mathsf{DTIME}[2^n] \subseteq \mathsf{E};$$

hence we have the strict inclusion $\mathsf{P} \subset \mathsf{E}$. Although we do not know whether $\mathsf{P} \subset \mathsf{NP}$, there exists a problem in E that cannot be solved in polynomial time. Other consequences of the Hierarchy Theorems are $\mathsf{NE} \subset \mathsf{NEXP}$ and $\mathsf{NL} \subset \mathsf{PSPACE}$.

In the Hierarchy Theorem for DTIME, the hypothesis on t_1 and t_2 is $t_1(n) \log t_1(n) = o(t_2(n))$, instead of $t_1(n) = o(t_2(n))$, for technical reasons related to the simulation of machines with multiple worktapes by a single universal Turing machine with a fixed number of worktapes. Other computational models, such as random access machines, enjoy tighter time hierarchy theorems.

All proofs of the Hierarchy Theorems use the technique of **diagonalization**. For example, the proof for DTIME constructs a Turing machine M of time complexity t_2 that considers all machines $M_1, M_2, \ldots$ whose time complexity is t_1; for each i, the proof finds a word x_i that is accepted by M if and only if $x_i \notin L(M_i)$, the language decided by M_i. Consequently, $L(M)$, the language decided by M, differs from each $L(M_i)$, hence $L(M) \notin \mathsf{DTIME}[t_1]$. The diagonalization technique resembles the classic method used to prove that the real numbers are uncountable, by constructing a number whose j^{th} digit differs from the j^{th} digit of the j^{th} number on the list. To illustrate the diagonalization technique, we outline the proof of the Hierarchy Theorem for DSPACE. In this subsection, $\langle i, x\rangle$ stands for the string $0^i 1x$, and $zeroes(y)$ stands for the number of 0's that a given string y starts with. Note that $zeroes(\langle i, x\rangle) = i$.

Proof. (of the DSPACE Hierarchy Theorem)

We construct a deterministic Turing machine M that decides a language A such that $A \in \mathsf{DSPACE}[s_2] - \mathsf{DSPACE}[s_1]$.

Let U be a deterministic universal Turing machine, as described in Section 2.3. On input x of length n, machine M performs the following:

1. Lay out $s_2(n)$ cells on a worktape.

2. Let $i = zeroes(x)$.

3. Simulate the universal machine U on input $\langle i, x \rangle$. Accept x if U tries to use more than s_2 worktape cells. (We omit some technical details, such as the way in which the constructibility of s_2 is used to ensure that this process halts.)

4. If U accepts $\langle i, x \rangle$, then reject; if U rejects $\langle i, x \rangle$, then accept.

Clearly, M always halts and uses space $O(s_2(n))$. Let $A = L(M)$.

Suppose $A \in \mathsf{DSPACE}[s_1(n)]$. Then there is some Turing machine M_j accepting A using space at most $s_1(n)$. Since the space used by U is $O(1)$ times the space used by M_j, there is a constant k depending only on j (in fact, we can take $k = |j|$), such that U, on inputs z of the form $z = \langle j, x \rangle$, uses at most $ks_1(|x|)$ space.

Since $s_1(n) = o(s_2(n))$, there is an n_0 such that $ks_1(n) \leq s_2(n)$ for all $n \geq n_0$. Let x be a string of length greater than n_0 such that the first $j+1$ symbols of x are 0^j1. Note that the universal Turing machine U, on input $\langle j, x \rangle$, simulates M_j on input x and uses space at most $ks_1(n) \leq s_2(n)$. Thus, when we consider the machine M defining A, we see that on input x the simulation does not stop in step 3, but continues on to step 4, and thus $x \in A$ if and only if U rejects $\langle j, x \rangle$. Consequently, M_j does not accept A, contrary to our assumption. Thus $A \notin \mathsf{DSPACE}[s_1(n)]$. □

Although the diagonalization technique successfully separates some pairs of complexity classes, diagonalization does not seem strong enough to separate P from NP. (See Theorem 6.1 below.)

4.5 Padding Arguments

A useful technique for establishing relationships between complexity classes is the **padding argument.** Let A be a language over alphabet Σ, and let # be a symbol not in Σ. Let f be a numeric function. The ***f*-padded version of** L is the language

$$A' = \{x\#^{f(n)} : x \in A \text{ and } n = |x|\}.$$

That is, each word of A' is a word in A concatenated with $f(n)$ consecutive # symbols. The padded version A' has the same information content as A, but because each word is longer, the computational complexity of A' is smaller.

The proof of the next theorem illustrates the use of a padding argument.

Theorem 4.5 *If* $\mathsf{P} = \mathsf{NP}$, *then* $\mathsf{E} = \mathsf{NE}$ [Book, 1974].

Proof. Since $\mathsf{E} \subseteq \mathsf{NE}$, we prove that $\mathsf{NE} \subseteq \mathsf{E}$.

Let $A \in \mathsf{NE}$ be decided by a nondeterministic Turing machine M in at most $t(n) = k^n$ time for some constant integer k. Let A' be the $t(n)$-padded version of A. From M, we construct a nondeterministic Turing machine M' that decides A' in linear time: M' checks that its input has the correct format, using the time-constructibility of t; then M' runs M on the prefix of the input preceding the first # symbol. Thus, $A' \in \mathsf{NP}$.

If $\mathsf{P} = \mathsf{NP}$, then there is a deterministic Turing machine D' that decides A' in at most $p'(n)$ time for some polynomial p'. From D', we construct a deterministic Turing machine D that decides A, as follows. On input x of length n, since $t(n)$ is time-constructible, machine D constructs $x\#^{t(n)}$, whose length is $n + t(n)$, in $O(t(n))$ time. Then D runs D' on this input word. The time complexity of D is at most $O(t(n)) + p'(n + t(n)) = 2^{O(n)}$. Therefore, $\mathsf{NE} \subseteq \mathsf{E}$. □

A similar argument shows that the $\mathsf{E} = \mathsf{NE}$ question is equivalent to the question of whether $\mathsf{NP} - \mathsf{P}$ contains a subset of 1^*, that is, a language over a single-letter alphabet.

5 Reducibility and Completeness

In this section, we discuss relationships between problems: informally, if one problem reduces to another problem, then in a sense, the second problem is harder than the first. The hardest problems in NP are the NP-complete problems. We define NP-completeness precisely, and we show how to prove that a problem is NP-complete. The theory of NP-completeness, together with the many known NP-complete problems, is perhaps the best justification for interest in the classes P and NP. All of the other canonical complexity classes listed above have natural and important problems that are complete for them; we give some of these as well.

5.1 Resource-Bounded Reducibilities

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. For example, the maximum matching problem for bipartite graphs reduces to the network flow problem (see Chapter *** CHAPTER REFERENCE: GRAPH AND NETWORK PROBLEMS ***). 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 computable 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., $A_1 = L(M^{A_2})$. (Computable functions and oracle Turing machines are defined in Section 2). 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 **log-space reducible** to 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 deterministically in $O(\log n)$ space.
- 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.) Since there are interesting distinctions to be made among problems in P, it is useful to have a more restrictive notion of reducibility.

Log-space reducibility [Jones, 1975] is just such a notion. Although it is not known whether whether log-space reducibility is different from Karp reducibility (just as it is not known whether L is a proper subset of P), this is widely believed to be true. (However, in practice, most problems that can be shown to be NP-complete under Karp reductions remain complete under log-space reductions.)

Theorem 5.1 *Log-space reducibility implies Karp reducibility, which implies Cook reducibility:*

1. *If* $A_1 \leq_m^{\log} A_2$, *then* $A_1 \leq_m^p A_2$.

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

Theorem 5.2 *Log-space reducibility, Karp reducibility, and Cook reducibility are transitive:*

1. *If* $A_1 \leq_m^{\log} A_2$ *and* $A_2 \leq_m^{\log} A_3$, *then* $A_1 \leq_m^{\log} A_3$.

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

3. *If* $A_1 \leq_T^p A_2$ *and* $A_2 \leq_T^p A_3$, *then* $A_1 \leq_T^p A_3$.

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 each input 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 (Theorem 5.3). 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 5.3

1. P *is closed under log-space reducibility, Karp reducibility, and Cook reducibility.*

2. NP *is closed under log-space reducibility and Karp reducibility.*

3. L *and* NL *are closed under log-space reducibility.*

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

5.2 Complete Languages

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.

Note, however, that some sources use 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 does *not* 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 5.3 in Section 5.1), 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 $\mathsf{co\text{-}NP}$, is that if $A_0 \in \mathsf{co\text{-}NP}$ then $\mathsf{NP} = \mathsf{co\text{-}NP}$. It is also believed unlikely that $\mathsf{NP} = \mathsf{co\text{-}NP}$, as was noted in Section 4.3 in connection with whether all tautologies have short proofs.

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 if $\mathsf{P} \neq \mathsf{NP}$. A theorem due to Ladner [1975] 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 $\mathsf{NTIME}[n^{10^{500}}]$.

5.3 Cook-Levin Theorem

Interest in NP-complete problems started with a theorem of Cook [1971], proved independently by Levin [1973]. Recall that SAT is the language of Boolean formulas $\phi(z_1, \ldots, z_r)$ such that there exists a truth assignment to the variables $z_1, \ldots, z_r$ that makes ϕ true.

Theorem 5.4 (Cook-Levin Theorem) SAT *is* NP*-complete.*

Proof. We know already that SAT is in NP, so to prove that SAT is NP-complete, we need to take an arbitrary given language A in NP and show that $A \leq_m^p$ SAT. Take N to be a nondeterministic Turing machine that decides A in polynomial time. Then the relation $R(x, y)$ = “y is a computation path of N

that leads it to accept x" is decidable in deterministic polynomial time depending only on $n = |x|$. We can assume that the length m of possible y's encoded as binary strings depends only on n and not on a particular x.

It is straightforward to show that there is a polynomial p and for each n a Boolean circuit C_n^R with $p(n)$ wires, with $n + m$ input wires labeled $x_1, \ldots, x_n, y_1, \ldots, y_m$ and one output wire w_0, such that $C_n^R(x, y)$ outputs 1 if and only if $R(x, y)$ holds. (We describe circuits in more detail below, and state a theorem for this principle as Part 1. of Theorem 9.1.) Importantly, C_n^R itself can be designed in time polynomial in n, and by the universality of NAND, may be composed entirely of binary NAND gates. Label the wires by variables $x_1, \ldots, x_n, y_1, \ldots, y_m, w_0, w_1, \ldots, w_{p(n)-n-m-1}$. These become the variables of our Boolean formulas. For each NAND gate g with input wires u and v, and for each output wire w of g, write down the subformula

$$\phi_{g,w} = (u \vee w) \wedge (v \vee w) \wedge (\bar{u} \vee \bar{v} \vee \bar{w}).$$

This subformula is satisfied by precisely those assignments to u, v, w that give $w = u \,\text{NAND}\, v$. The conjunction ϕ_0 of $\phi_{g,w}$ over the polynomially many gates g and their output wires w thus is satisfied only by assignments that set every gate's output correctly given its inputs. Thus for any binary strings x and y of lengths n, m respectively, the formula $\phi_1 = \phi_0 \wedge w_0$ is satisfiable by a setting of the wire variables $w_0, w_1, \ldots, w_{p(n)-n-m-1}$ if and only if $C_n^R(x, y) = 1$ – i.e., if and only if $R(x, y)$ holds.

Now given any fixed x and taking $n = |x|$, the Karp reduction computes ϕ_1 as above, and finally outputs the Boolean formula ϕ obtained by substituting the bit-values of x into ϕ_1. This ϕ has variables $y_1, \ldots, y_m, w_0, w_1, \ldots, w_{p(n)-n-m-1}$, and the computation of ϕ from x runs in deterministic polynomial time. Then $x \in A$ if and only if N accepts x, if and only if there exists y such that $R(x, y)$ holds, if and only if there exists an assignment to the variables $w_0, w_1, \ldots, w_{p(n)-n-m-1}$ *and* $y_1, \ldots, y_m$ that satisfies ϕ, if and only if $\phi \in$ SAT. This shows $A \leq_m^p$ SAT. □

We have actually proved that SAT remains NP-complete even when the given instances ϕ are *restricted* to Boolean formulas that are a conjunction of *clauses*, where each clause consists of (here, at most three) disjuncted literals. Such formulas are said to be in *conjunctive normal form*. Theorem 5.4 is also commonly known as **Cook's Theorem**.

5.4 Proving NP-Completeness

After one language has been proved complete for a class, others can be proved complete by constructing transformations. For NP, if A_0 is NP-complete, then to prove that another language A_1 is NP-complete, it suffices to prove that $A_1 \in$ 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 5.2), $A \leq_m^p A_1$.

Beginning with Cook [1971] and Karp [1972], 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. The following NP-complete decision problems are frequently used in these reductions—the language corresponding to each problem is the set of instances whose answers are `yes`.

- 3-SATISFIABILITY (3SAT)

  Instance: A Boolean expression ϕ in conjunctive normal form with three literals per clause [e.g., $(w \vee x \vee \overline{y}) \wedge (\overline{x} \vee y \vee z)$]

  Question: Is ϕ satisfiable?

- 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 of W?

- CLIQUE

  Instance: A graph G and an integer k

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

- 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?

- HAMILTONIAN CIRCUIT

  Instance: A graph G

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

- THREE-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 $S - S'$?

Note that our ϕ in the above proof of the Cook-Levin Theorem already meets a form of the definition of 3SAT relaxed to allow "at most 3 literals per clause." Padding ϕ with some extra variables to bring up the number in each clause to exactly three, while preserving whether the formula is satisfiable or not, is not difficult, and establishes the NP-completeness of 3SAT. Below we use this in showing the NP-completeness of INDEPENDENT SET.

Some NP-completeness proofs require only "minor adjustments" even between different-looking problems. 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 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.

Here is another example of an NP-completeness proof, for the following decision problem:

- TRAVELING SALESMAN PROBLEM (TSP)

  Instance: A set of m "cities" $C_1, \ldots, C_m$, with an integer 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

$$d(c_1, c_2) + \cdots + d(c_{m-1}, c_m) + d(c_m, c_1) \leq D?$$

First, it is easy to see that TSP is in NP: a nondeterministic Turing machine simply guesses a tour and checks that the total length is at most D.

Next, we construct a reduction from Hamiltonian Circuit to TSP. (The reduction goes from the known NP-complete problem, Hamiltonian Circuit, to the new problem, TSP, not vice versa.)

From a graph G on m vertices $v_1, \ldots, v_m$, define the 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. Each vertex of G corresponds to a city in the constructed instance of TSP.

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 $m+1$. Thus, each step corresponds to an edge of G, and the corresponding sequence of vertices in G is a Hamiltonian circuit.

5.5 NP-Completeness by Combinatorial Transformation

The following example shows 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 5.5 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 3.

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.

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. □

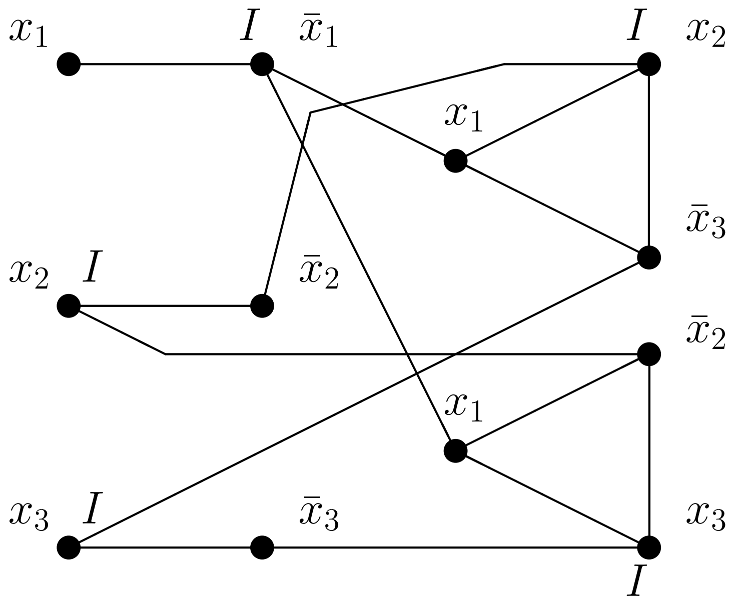

Figure 3: Construction in the proof of NP-completeness of INDEPENDENT SET for the formula $(x_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3)$. The independent set of size 5 corresponding to the satisfying assignment $x_1 =$ `false`, $x_2 =$ `true`, and $x_3 =$ `true` is shown by nodes marked I.

5.6 Complete Problems for Other Classes

Besides NP, the following canonical complexity classes have natural complete problems. The three problems now listed are complete for their respective classes under log-space reducibility.

- NL: GRAPH ACCESSIBILITY PROBLEM

  Instance: A directed graph G with nodes $1, \ldots, N$

  Question: Does G have a directed path from node 1 to node N?

- P: CIRCUIT VALUE PROBLEM

  Instance: A Boolean circuit (see Section 9) with output node u, and an assignment I of $\{\mathbf{0,1}\}$ to each input node

  Question: Is $\mathbf{1}$ the value of u under I?

- PSPACE: QUANTIFIED BOOLEAN FORMULAS

  Instance: A Boolean expression with all variables quantified with either $\forall$ or $\exists$ [e.g., $\forall x \forall y \exists z(x \wedge (\overline{y} \vee z))$].

  Question: Is the expression `true`?

These problems can be used to prove other problems are NL-complete, P-complete, and PSPACE-complete, respectively.

Stockmeyer and Meyer [1973] defined a natural decision problem that they proved to be complete for NE. If this problem were in P, then by closure under Karp reducibility (Theorem 5.3), we would have

$\mathsf{NE} \subseteq \mathsf{P}$, a contradiction of the hierarchy theorems (Theorem 4.4). 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. Although diagonalization produces unnatural problems by itself, the combination of diagonalization and completeness shows that *natural* problems are intractable.

The next section points out some limitations of current diagonalization techniques.

6 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 the 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 6.1 *There exist languages A and B such that $\mathsf{P}^A = \mathsf{NP}^A$, and $\mathsf{P}^B \neq \mathsf{NP}^B$.* [Baker et al., 1975].

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. Important non-relativizing proof techniques were discovered in the early 1990s, in connection with interactive proof systems (Section 12.1).

7 The Polynomial Hierarchy

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

- $\mathsf{NP}^{\mathcal{C}} = \bigcup_{A \in \mathcal{C}} \mathsf{NP}^A$,
- $\Sigma_0^P = \Pi_0^P = \mathsf{P}$;

and for $k \geq 0$, define

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

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, $\Sigma_k^P \cup \Pi_k^P \subseteq \Sigma_{k+1}^P \cap \Pi_{k+1}^P$, but this inclusion is not known to be strict.

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. In the next section, we define the polynomial hierarchy in two other ways, one of which is in terms of alternating Turing machines.

8 Alternating Complexity Classes

In this section, we define time and space complexity classes for alternating Turing machines, and we show how these classes are related to the classes introduced already. The possible computations of an alternating Turing machine M on an input word x can be represented by a tree T_x in which the root is the initial configuration, and the children of a nonterminal node C are the configurations reachable from C by one step of M. For a word x in $L(M)$, define an **accepting subtree** S of T_x to be a subtree of T_x with the following properties:

- S is finite.
- The root of S is the initial configuration with input word x.
- If S has an existential configuration C, then S has exactly one child of C in T_x; if S has a universal configuration C, then S has all children of C in T_x.
- Every leaf is a configuration whose state is the accepting state q_A.

Observe that each node in S is an accepting configuration.

We consider only alternating Turing machines that always halt. For $x \in L(M)$, define the time taken by M to be the height of the shortest accepting tree for x, and the space to be the maximum number of worktape cells visited along any path in an accepting tree that minimizes this number. For $x \notin L(M)$, define the time to be the height of T_x, and the space to be the maximum number of worktape cells visited along any path in T_x.

Let $t(n)$ be a time-constructible function, and let $s(n)$ be a space-constructible function. Define the following complexity classes:

- $\mathsf{ATIME}[t(n)]$ is the class of languages decided by alternating Turing machines of time complexity $O(t(n))$.
- $\mathsf{ASPACE}[s(n)]$ is the class of languages decided by alternating Turing machines of space complexity $O(s(n))$.

Because a nondeterministic Turing machine is a special case of an alternating Turing machine, for every $t(n)$ and $s(n)$, $\mathsf{NTIME}[t] \subseteq \mathsf{ATIME}[t]$ and $\mathsf{NSPACE}[s] \subseteq \mathsf{ASPACE}[s]$. The next theorem states further relationships between computational resources used by alternating Turing machines, and resources used by deterministic and nondeterministic Turing machines.

Theorem 8.1 (Alternation Theorems) [Chandra et al., 1981]. *Let $t(n)$ be a time-constructible function, and let $s(n)$ be a space-constructible function, $s(n) \geq \log n$.*

(a) $\mathsf{NSPACE}[s(n)] \subseteq \mathsf{ATIME}[s(n)^2]$

(b) $\mathsf{ATIME}[t(n)] \subseteq \mathsf{DSPACE}[t(n)]$

(c) $\mathsf{ASPACE}[s(n)] \subseteq \mathsf{DTIME}[2^{O(s(n))}]$

(d) $\mathsf{DTIME}[t(n)] \subseteq \mathsf{ASPACE}[\log t(n)]$

In other words, space on deterministic and nondeterministic Turing machines is polynomially related to time on alternating Turing machines. Space on alternating Turing machines is exponentially related to time on deterministic Turing machines. The following corollary is immediate.

Theorem 8.2

(a) $\mathsf{ASPACE}[O(\log n)] = \mathsf{P}$.

(b) $\mathsf{ATIME}[n^{O(1)}] = \mathsf{PSPACE}$.

(c) $\mathsf{ASPACE}[n^{O(1)}] = \mathsf{EXP}$.

In Section 7, we defined the classes of the polynomial hierarchy in terms of oracles, but we can also define them in terms of alternating Turing machines with restrictions on the number of alternations between existential and universal states. 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.

Theorem 8.3 [Stockmeyer, 1976; Wrathall, 1976]. *For any language A, the following are equivalent:*

1. $A \in \Sigma_k^P$.
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 P and a polynomial p such that for all x, $x \in A$ if and only if*

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

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

Alternating Turing machines are closely related to Boolean circuits, which are defined in the next section.

9 Circuit Complexity

The hardware of electronic digital computers is based on digital logic gates, connected into combinational circuits. (See Chapter ***CHAPTER REFERENCE: DIGITAL LOGIC***.) Here, we specify a model of computation that formalizes the combinational circuit.

A *Boolean circuit* on n input variables $x_1, \ldots, x_n$ is a directed acyclic graph with exactly n input nodes of indegree 0 labeled $x_1, \ldots, x_n$, and other nodes of indegree 1 or 2, called *gates*, labeled with the Boolean operators in $\{\wedge, \vee, \neg\}$. One node is designated as the output of the circuit. See Fig. 4. Without loss of generality, we assume that there are no extraneous nodes; there is a directed path from each node to the output node. The indegree of a gate is also called its *fan-in*.

An *input assignment* a function I that maps each variable x_i to either 0 or 1. The value of each gate g under I is obtained by applying the Boolean operation that labels g to the values of the immediate predecessors of g. The function computed by the circuit is the value of the output node for each input assignment.

A Boolean circuit computes a finite function: a function of only n binary input variables. To decide membership in a language, we need a circuit for each input length n.

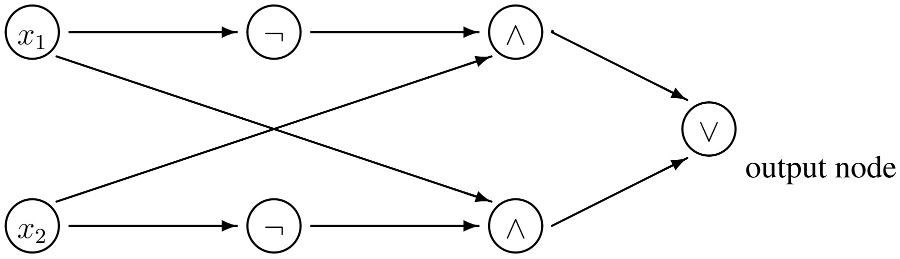

Figure 4: A Boolean circuit.

A *circuit family* is an infinite set of circuits $C = \{c_1, c_2, \ldots\}$ in which each c_n is a Boolean circuit on n inputs. C *decides* a language $A \subseteq \{\mathbf{0,1}\}^*$ if for every n and every assignment $a_1, \ldots, a_n$ of $\{\mathbf{0,1}\}$ to the n inputs, the value of the output node of c_n is $\mathbf{1}$ if and only if the word $a_1 \cdots a_n \in A$. The *size complexity* of C is the function $z(n)$ that specifies the number of nodes in each c_n. The *depth complexity* of C is the function $d(n)$ that specifies the length of the longest directed path in c_n. Clearly, since the fan-in of each gate is at most 2, $d(n) \geq \log z(n) \geq \log n$. The class of languages decided by polynomial-size circuits is denoted by $\mathsf{P/poly}$.

With a different circuit for each input length, a circuit family could solve an undecidable problem such as the halting problem (see Chapter *** CHAPTER REFERENCE: FORMAL MODELS AND COMPUTABILITY ***). For each input length, a table of all answers for machine descriptions of that length could be encoded into the circuit. Thus, we need to restrict our circuit families. The most natural restriction is that all circuits in a family should have a concise, uniform description, to disallow a different answer table for each input length. Several uniformity conditions have been studied; the following is the easiest to present:

A circuit family $\{c_1, c_2, \ldots\}$ of size complexity $z(n)$ is *log-space uniform* if there exists a deterministic Turing machine M such that on each input of length n, machine M produces a description of c_n, using space $O(\log z(n))$.

Now we define complexity classes for uniform circuit families and relate these classes to previously defined classes. Define the following complexity classes:

- $\mathsf{SIZE}[z(n)]$ is the class of languages decided by log-space uniform circuit families of size complexity $O(z(n))$.

- $\mathsf{DEPTH}[d(n)]$ is the class of languages decided by log-space uniform circuit families of depth complexity $O(d(n))$.

In our notation, $\mathsf{SIZE}[n^{O(1)}]$ equals P, which is a proper subclass of $\mathsf{P/poly}$.

Theorem 9.1

1. *If $t(n)$ is a time-constructible function, then* $\mathsf{DTIME}[t(n)] \subseteq \mathsf{SIZE}[t(n) \log t(n)]$ [Pippenger and Fischer, 1979].

2. $\mathsf{SIZE}[z(n)] \subseteq \mathsf{DTIME}[z(n)^{O(1)}]$.

3. *If $s(n)$ is a space-constructible function and $s(n) \geq \log n$, then* $\mathsf{NSPACE}[s(n)] \subseteq \mathsf{DEPTH}[s(n)^2]$ [Borodin, 1977].

4. *If $d(n) \geq \log n$, then* $\mathsf{DEPTH}[d(n)] \subseteq \mathsf{DSPACE}[d(n)]$ [Borodin, 1977].

The next theorem shows that size and depth on Boolean circuits are closely related to space and time on alternating Turing machines, provided that we permit sublinear running times for alternating Turing machines, as follows. We augment alternating Turing machines with a random-access input capability. To access the cell at position j on the input tape, M writes the binary representation of j on a special tape, in $\log j$ steps, and enters a special reading state to obtain the symbol in cell j.

Theorem 9.2 [Ruzzo, 1979]. *Let $t(n) \geq \log n$ and $s(n) \geq \log n$ be such that the mapping $n \mapsto (t(n), s(n))$ (in binary) is computable in time $s(n)$.*

1. *Every language decided by an alternating Turing machine of simultaneous space complexity $s(n)$ and time complexity $t(n)$ can be decided by a log-space uniform circuit family of simultaneous size complexity $2^{O(s(n))}$ and depth complexity $O(t(n))$.*

2. *If $d(n) \geq (\log z(n))^2$, then every language decided by a log-space uniform circuit family of simultaneous size complexity $z(n)$ and depth complexity $d(n)$ can be decided by an alternating Turing machine of simultaneous space complexity $O(\log z(n))$ and time complexity $O(d(n))$.*

The correspondence between time on alternating Turing machines and depth on log-space uniform circuits breaks down for time and depth less than $\log^2 n$, primarily because L is not known to be contained in $\mathsf{ATIME}(o(\log^2 n))$. This has motivated the adoption of more restrictive notions of uniformity (see, e.g., [Ruzzo, 1981], [Vollmer, 1999]). When researchers mention small uniform circuit classes, they usually are referring to these more restrictive notions of uniformity. (For depths at least $\log n$, the complexity classes obtained via these different notions of uniformity coincide.)

In a sense, the Boolean circuit family is a model of parallel computation, because all gates compute independently, in parallel. For each $k \geq 0$, NC^k denotes the class of languages decided by uniform bounded-fan-in circuits of polynomial size and depth $O((\log n)^k)$, and AC^k is defined analogously for unbounded fan-in circuits. In particular, AC^k is the same as the class of languages decided by a parallel machine model called the CRCW PRAM with polynomially many processors in parallel time $O((\log n)^k)$ [Stockmeyer and Vishkin, 1984].

10 Probabilistic Complexity Classes

Since the 1970s, with the development of randomized algorithms for computational problems (see Chapter *** CHAPTER REFERENCE: RANDOMIZED ALGORITHMS ***), 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 positive 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 primality testing algorithm of Solovay and Strassen [1977], 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 has also sometimes been 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 RP $\subseteq$ BPP.)

Theorem 10.1 *If $A \in$ 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 $x \in A$, and $p_M(x) < 1/2^{q(n)}$ for every $x \notin A$.*

If $L \in$ 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

- ZPP $=$ RP $\cap$ co-RP.

The letter Z in ZPP is for zero probability of error, as we now demonstrate. Suppose $A \in$ 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.

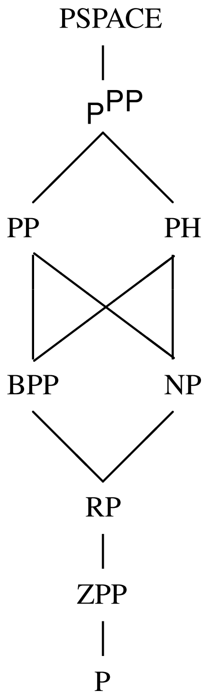

Figure 5: Probabilistic complexity classes.

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 7 and Figure 10.

Theorem 10.2

(a) $\mathsf{P} \subseteq \mathsf{ZPP} \subseteq \mathsf{RP} \subseteq \mathsf{BPP} \subseteq \mathsf{PP} \subseteq \mathsf{PSPACE}$ [Gill, 1977].

(b) $\mathsf{RP} \subseteq \mathsf{NP} \subseteq \mathsf{PP}$ [Gill, 1977].

(c) $\mathsf{BPP} \subseteq \Sigma_2^P \cap \Pi_2^P$ [Lautemann, 1983; Sipser, 1983].

(d) $\mathsf{BPP} \subset \mathsf{P/poly}$.

(e) $\mathsf{PH} \subseteq \mathsf{P}^{\mathsf{PP}}$ [Toda, 1991].

An important recent research area called **de-randomization** studies whether randomized algorithms can be converted to deterministic ones of the same or comparable efficiency. For example, if there is a language in E that requires Boolean circuits of size $2^{\Omega(n)}$ to decide it, then $\mathsf{BPP} = \mathsf{P}$ [Impagliazzo and Wigderson, 1997].

11 Quantum Computation

A probabilistic computer enables one to sample efficiently over exponentially many possible computation paths. However, if $\mathsf{BPP} = \mathsf{P}$ then the added capability is weak. A *quantum computer*, however, can harness parallelism and randomness in ways that appear theoretically to be much more powerful, ways that are able to solve problems believed not to lie in BPP. A key difference is that by quantum *interference*, opening up new computation paths may cause the probabilities of some other paths to *vanish*, whereas for a BPP-machine those probabilities would remain positive.

- BQP is the class of languages decided by quantum machines of polynomial time complexity with bounded two-sided error.

A prime motivation for studying BQP is that it includes language versions of the integer factoring and discrete logarithm problems, which are not known to belong to BPP. The public-key cryptosystems in common use today rely on the presumed intractability of these problems, and are theoretically breakable by eavesdroppers armed with quantum computers. This is one of several reasons why the interplay of quantum mechanics and computational complexity is important to cryptographers.

In terms of the complexity classes depicted in Figure 5, the only inclusions that are known involving BQP are

$$\mathsf{BPP} \subseteq \mathsf{BQP} \subseteq \mathsf{PP}.$$

In particular, it is important to emphasize that it is *not* generally believed that quantum computers can solve NP-complete problems quickly. For more information on quantum computation, see Chapter *** CHAPTER REFERENCE: QUANTUM COMPUTATION ***).

12 Interactive Models and Complexity Classes

12.1 Interactive Proofs

In Section 3.2, 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** [Goldwasser et al., 1989], 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 [Babai and Moran, 1988].

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

- A read-only input tape on which an input string x is written.
- 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 *prover* P, which receives messages from V and sends messages to V.
- 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.

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.

It is important to comment that de-randomization techniques applied to these proof systems have shown that under plausible hardness assumptions, proofs of non-isomorphism of sub-exponential length (or even polynomial length) do exist [Klivans and van Melkebeek, 2002]. Thus many complexity theoreticians now conjecture that the graph isomorphism problem lies in NP $\cap$ co-NP.

The complexity class IP comprises the languages A for which there exists a verifier V and a positive 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 8.1), it is straightforward to show that IP $\subseteq$ PSPACE. It was originally believed likely that IP was a small subclass of PSPACE. Evidence supporting this belief was the construction of an oracle language B for which co-$\mathsf{NP}^B - \mathsf{IP}^B \neq \emptyset$ [Fortnow and Sipser, 1988], so that IP^B is strictly included in PSPACE^B. Using a proof technique that does not relativize, however, Shamir [1992] proved that in fact, IP and PSPACE are the same class.

Theorem 12.1 IP $=$ PSPACE. [Shamir, 1992].

If NP is a proper subset of PSPACE, as is widely believed, then Theorem 12.1 says that interactive proof systems can decide a larger class of languages than NP.

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

- for all $x \in A$, there exists an oracle language B_x such that M^{B_x} accepts x with probability 1.
- for all $x \notin A$, and for every language B, machine M^B accepts x with probability strictly less than $1/2$.

Intuitively, the oracle language B_x represents a proof of membership of x in A. 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 12.2 *A has a probabilistically checkable proof if and only if* $A \in \mathsf{NEXP}$ [Babai et al., 1991].

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 $r(n)$ random binary choices, and queries its oracle $q(n)$ 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 12.3 (The PCP Theorem) $\mathsf{NP} = \mathsf{PCP}[O(\log n), O(1)]$ [Arora et al., 1998].

Theorem 12.3 asserts that for every language A in NP, a proof that $x \in A$ 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.

13 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.

Our goal will be to equate the complexity of a string with the length of its shortest "description" – but this raises the question of what class of "descriptions" should be considered. Given any Turing machine M, one can define a measure $C_M(x)$ to be the length of the shortest input d (the "description") such that $M(d) = x$. At first, this might not seem to be a very promising approach, because using different machines M_1 and M_2 will yield measures C_{M_1} and C_{M_2} that differ wildly (or worse, are undefined for those x that are never produced as output). Fortunately, the next theorem shows that if one chooses a *universal* Turing machine U to define a measure $C_U(x)$ in this way, the resulting measure is roughly the same as that obtained by using any other universal machine.

Theorem 13.1 (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, $C_U(y) \leq C_{U'}(y) + c$.*

Henceforth, let C be defined by the universal Turing machine of Theorem 13.1. For every integer n and every binary string y of length n, because y can be described by giving itself explicitly, $C(y) \leq n + c'$ for a constant c'. Call y **incompressible** if $C(y) \geq n$. 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.

14 Research Issues and Summary

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 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.

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 has provided 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). All of these systems are thus based on wishful thinking and conjecture. Research is needed to resolve these open questions and replace conjecture with mathematical certainty.

Acknowledgments

Donna Brown, Bevan Das, Raymond Greenlaw, Lane Hemaspaandra, John Jozwiak, Sung-il Pae, Leonard Pitt, Michael Roman, and Martin Tompa read earlier versions of this chapter and suggested numerous helpful improvements. Karen Walny checked the references.

Defining Terms

Complexity class: A set of languages that are decided within a particular resource bound. For example, $\mathsf{NTIME}(n^2 \log n)$ is the set of languages decided by nondeterministic Turing machines within $O(n^2 \log n)$ time.

Constructibility: A function $f(n)$ is time (respectively, space) constructible if there exists a deterministic Turing machine that halts after exactly $f(n)$ steps [after using exactly $f(n)$ worktape cells] for every input of length n.

Diagonalization: A technique for constructing a language A that differs from every $L(M_i)$ for a list of machines $M_1, M_2, \ldots$.

NP-complete: A language A_0 is NP-complete if $A_0 \in \mathsf{NP}$ and $A \leq_m^p A_0$ for every A in NP; that is, for every A in NP, there exists a function f computable in polynomial time such that for every x, $x \in A$ if and only if $f(x) \in A_0$.

Oracle: An oracle is a language A to which a machine presents queries of the form "Is w in A" and receives each correct answer in one step.

Padding: A technique for establishing relationships between complexity classes that uses padded versions of languages, in which each word is padded out with multiple occurrences of a new symbol—the word x is replaced by the word $x\#^{f(|x|)}$ for a numeric function f—in order to artificially lower the complexity of the language.

Reduction: A language A reduces to a language B if a machine that decides B can be used to decide A efficiently.

Time and space complexity: The time (respectively, space) complexity of a deterministic Turing machine M is the maximum number of steps taken (worktape cells visited) by M among all input words of length n.

Turing machine: A Turing machine M is a model of computation with a read-only input tape and multiple worktapes. At each step, M reads the tape cells on which its access heads are located, and depending on its current state and the symbols in those cells, M changes state, writes new symbols on the worktape cells, and moves each access head one cell left or right or not at all.

References

[1] Allender, E. 2009. A status report on the P versus NP question. *Advances in Computers* 77:118–147.

[2] Allender, E., Loui, M.C., and Regan, K.W. 1999. Chapter 23: Complexity classes, Chapter 24: Reducibility and completeness, Chapter 25: Other complexity classes and measures. In *Algorithms and Theory of Computation Handbook, Second Edition, Volume 1: General Concepts and Techniques*. Chapman and Hall/CRC, 2009.

[3] Arora, S., Lund, C., Motwani, R., Sudan, M., and Szegedy, M. 1998. Proof verification and hardness of approximation problems. *J. ACM* 45(3):501–555.

[4] Arora, S., and Barak, B. 2009. *Complexity Theory: A Modern Approach.* Cambridge University Press, Cambridge, UK.

[5] Babai, L. and Moran, S. 1988. Arthur-Merlin games: a randomized proof system, and a hierarchy of complexity classes. *J. Comput. Sys. Sci.* 36(2):254–276.

[6] Babai, L., Fortnow, L., and Lund, C. 1991. Nondeterministic exponential time has two-prover interactive protocols. *Computational Complexity*, 1:3–40.

[7] Baker, T., Gill, J., and Solovay, R. 1975. Relativizations of the P=NP? question. *SIAM J. Comput.*, 4(4):431–442.

[8] Book, R.V. 1974. Comparing complexity classes. *J. Comp. Sys. Sci.*, 9(2):213–229.

[9] Borodin, A. 1972. Computational complexity and the existence of complexity gaps. *J. Assn. Comp. Mach.*, 19(1):158–174.

[10] Borodin, A. 1977. On relating time and space to size and depth. *SIAM J. Comput.*, 6(4):733–744.

[11] Chandra, A. K., Kozen, D. C., and Stockmeyer, L. J. 1981. Alternation. *J. Assn. Comp. Mach.* 28(1):114–133.

[12] Cook, S.A. 1971. The complexity of theorem-proving procedures. In *Proc. 3rd Annu. ACM Symp. Theory Comput.*, pp. 151–158. Shaker Heights, OH.

[13] Fortnow, L. 2009. The history and status of the P vs. NP problem. *Communications of the ACM*, 52(9):78–86.

[14] Fortnow, L. and Sipser, M. 1988. Are there interactive protocols for co-NP languages? *Inform. Process. Lett.* 28(5):249–251.

[15] Garey, M.R. and Johnson, D.S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman, San Francisco.

[16] Gill, J. 1977. Computational complexity of probabilistic Turing machines. *SIAM J. Comput.*, 6(4):675–695.

[17] Goldreich, O. 2008. *Computational Complexity: A Conceptual Perspective*. Cambridge University Press, Cambridge, UK.

[18] Goldwasser, S., Micali, S., and Rackoff, C. 1989. The knowledge complexity of interactive proof systems. *SIAM J. Comput.* 18(1):186–208.

[19] Hartmanis, J., ed. 1989. *Computational Complexity Theory.* American Mathematical Society, Providence, RI.

[20] Hartmanis, J. and Stearns, R. E. 1965. On the computational complexity of algorithms. *Trans. Amer. Math. Soc.* 117:285–306.

[21] Hemaspaandra, L.A. and Ogihara, M. 2002. *The Complexity Theory Companion*. Springer-Verlag, Berlin/New York.

[22] Hennie, F. and Stearns, R.A. 1966. Two–way simulation of multitape Turing machines. *J. Assn. Comp. Mach.* 13(4):533–546.

[23] Homer, S., and Selman, A. 2011. *Computability and Complexity Theory (2nd ed.)*. Springer-Verlag, Berlin/New York.

[24] Hopcroft, J., Motwani, R., and Ullman, J. 2000. *Introduction to Automata Theory, Languages, and Computation (2nd. ed.)*, Addison-Wesley, Reading, MA.

[25] Immerman, N. 1988. Nondeterministic space is closed under complementation. *SIAM J. Comput.*, 17(5):935–938.

[26] Impagliazzo, R. and Wigderson, A. 1997. P = BPP if E requires exponential circuits: Derandomizing the XOR lemma. *Proc. 29th Annu. ACM Symp. Theory Comput.*, ACM Press, pp 220–229. El Paso, Texas.

[27] Jones, N.D. 1975. Space-bounded reducibility among combinatorial problems. *J. Comp. Sys. Sci.*, 11(1):68–85. Corrigendum *J. Comp. Sys. Sci.* 15(2):241, 1977.

[28] Karp, R.M. 1972. Reducibility among combinatorial problems. In *Complexity of Computer Computations.* R.E. Miller and J.W. Thatcher, eds., pp. 85–103. Plenum Press, New York.

[29] Klivans, A.R. and van Melkebeek, D. 2002. Graph nonisomorphism has subexponential size proofs unless the polynomial-time hierarchy collapses. *SIAM J. Comput.* 31(5):1501–1526.

[30] Ladner, R.E. 1975. On the structure of polynomial-time reducibility. *J. Assn. Comp. Mach.* 22(1):155–171.

[31] Lautemann, C. 1983. BPP and the polynomial hierarchy. *Inf. Proc. Lett.*, 17(4):215–217.

[32] Li, M. and Vitányi, P.M.B. 2008. *An Introduction to Kolmogorov Complexity and Its Applications*. 3rd ed. Springer–Verlag, Berlin/New York.

[33] Moore, C., and Mertens, S. 2011. *The Nature of Computation*. Oxford University Press, Oxford, UK.

[34] Papadimitriou, C.H. 1994. *Computational Complexity*. Addison–Wesley, Reading, MA.

[35] Pippenger, N., and Fischer, M. 1979. Relations among complexity measures. *J. Assn. Comp. Mach.*, 26(2):361–381.

[36] Ruzzo, W.L. 1981. On uniform circuit complexity. *J. Comp. Sys. Sci.*, 22(3):365–383.

[37] Savitch, W.J. 1970. Relationship between nondeterministic and deterministic tape complexities. *J. Comp. Sys. Sci.*, 4(2):177–192.

[38] Seiferas, J.I., Fischer, M.J., and Meyer, A.R. 1978. Separating nondeterministic time complexity classes. *J. Assn. Comp. Mach.*, 25(1):146–167.

[39] Shamir, A. 1992. IP = PSPACE. *J. ACM* 39(4):869–877.

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

[41] Sipser, M. 1992. The history and status of the P versus NP question. In *Proc. 24th Annu. ACM Symp. Theory Comput.*, ACM Press, pp. 603–618. Victoria, B.C., Canada.

[42] Sipser, M. 2005. Introduction to the Theory of Computation. Thomson, Boston MA.

[43] Solovay, R., and Strassen, V. 1977. A fast Monte-Carlo test for primaility. *SIAM J. Comput.* 6(1):84–85.

[44] Stearns, R.E. 1990. Juris Hartmanis: the beginnings of computational complexity. In *Complexity Theory Retrospective.* A.L. Selman, ed., pp. 5–18, Springer-Verlag, Berlin/New York.

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

[46] Stockmeyer, L.J. and Chandra, A.K. 1979. Intrinsically difficult problems. *Sci. Am.* 240(5):140–159.

[47] Stockmeyer, L.J. and Meyer, A.R. 1973. Word problems requiring exponential time: preliminary report. In *Proc. 5th Annu. ACM Symp. Theory Comput.*, ACM Press, pp. 1–9. Austin, TX.

[48] Stockmeyer, L.J. and Vishkin, U. 1984. Simulation of parallel random access machines by circuits. *SIAM J. Comput.* 13(2):409–422.

[49] Szelepcsényi, R. 1988. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26(3):279–284.

[50] Toda, S. 1991. PP is as hard as the polynomial-time hierarchy. *SIAM J. Comput.*, 20(5):865–877.

[51] van Leeuwen, J. 1990. *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*. Elsevier Science, Amsterdam, and M.I.T. Press, Cambridge, MA.

[52] Wigderson, A. 2006. P, NP, and mathematics—a computational complexity perspective. In *Proceedings of the International Congress of Mathematicians, Madrid, vol. 1*, EMS Publishing House, pp 665–712.

[53] Wrathall, C. 1976. Complete sets and the polynomial-time hierarchy. *Theor. Comp. Sci.*, 3(1):23–33.

Further Information

This chapter is a short version of three chapters written by the same authors for the *Algorithms and Theory of Computation Handbook* [Allender et al., 1999].

The formal theoretical study of computational complexity began with the paper of Hartmanis and Stearns [1965], who introduced the basic concepts and proved the first results. For historical perspectives on complexity theory, see Hartmanis [1994], Wigderson [2006], Allender [2009], and Fortnow [2009].

Contemporary textbooks on complexity theory are by Moore and Mertens [2011], Homer and Selman [2011], Arora and Barak [2009], Goldreich [2008], while venerable texts include Sipser [2005], Hopcroft, Ullman, and Motwani [2000], Hemaspaandra and Ogihara [2002], and Papadimitriou [1994]. A good general reference is the *Handbook of Theoretical Computer Science* [van Leeuwen, 1990], volume A.

The text by Garey and Johnson [1979] explains NP-completeness thoroughly, with examples of NP-completeness proofs, and a collection of hundreds of NP-complete problems. Li and Vitányi [2008] provide a comprehensive scholarly treatment of Kolmogorov complexity, with many applications. One source for surveys and lecture notes on complexity theory is `http://web.ift.uib.no/~antonych/scripts.html`, by A. Czumaj and M. Kutylowski.

Five organizations and mainstay websites that promote research in computational complexity online sources are the the ACM Special Interest Group on Algorithms and Computation Theory (SIGACT), the European Association for Theoretical Computer Science (EATCS), the annual IEEE Conference on Computational Complexity, the "Complexity Zoo," and the theory section of `StackExchange`:

```
http://sigact.org/
http://www.eatcs.org/
http://www.computationalcomplexity.org/
http://www.complexityzoo.com/
http://cstheory.stackexchange.com/
```

The *Electronic Colloquium on Computational Complexity* maintained at the University of Trier, Germany at `http://eccc.hpi-web.de/`, and `arxiv.org/` run by Cornell University, include downloadable current research papers in the field, often with updates and revisions. Most individual researchers have websites with their work, and some also write blogs where news and ideas can be found. Among the most prominent blogs are:

Computational Complexity, `http://blog.computationalcomplexity.org/`, by Lance Fortnow and William Gasarch.

Shtetl-Optimized, `http://www.scottaaronson.com/blog/`, by Scott Aaronson.

Gödel's Lost Letter and P=NP, `http://rjlipton.wordpress.com/`, by Richard Lipton and the third author of this chapter.

Research papers on complexity theory are presented at several annual conferences, including the ACM Symposium on Theory of Computing (STOC); the IEEE Symposium on Foundations of Computer Science (FOCS), the International Colloquium on Automata, Languages, and Programming (ICALP) sponsored by EATCS, and the Symposium on Theoretical Aspects of Computer Science (STACS). The annual Conference on Computational Complexity (formerly Structure in Complexity Theory), also sponsored by the IEEE, is entirely devoted to complexity theory. Journals in which complexity research often appears include: *Chicago Journal on Theoretical Computer Science, Computational Complexity, Information and Computation, Journal of the ACM, Journal of Computer and System Sciences, SIAM Journal on Computing, Theoretical Computer Science, Theory of Computing Systems* (formerly *Mathematical Systems Theory*), *ACM Transactions on Computation Theory*, and *Theory of Computing*. Each issue of *ACM SIGACT News* and *Bulletin of the EATCS* contains a column on complexity theory.