Introduction to Computational Complexity[*]

— Mathematical Programming Glossary Supplement —

Magnus Roos and Jörg Rothe
Institut für Informatik
Heinrich-Heine-Universität Düsseldorf
40225 Düsseldorf, Germany
{roos,rothe}@cs.uni-duesseldorf.de

March 24, 2010

Abstract

This supplement is a brief introduction to the theory of computational complexity, which in particular provides important notions, techniques, and results to classify problems in terms of their complexity. We describe the foundations of complexity theory, survey upper bounds on the time complexity of selected problems, define the notion of polynomial-time many-one reducibility as a means to obtain lower bound (i.e., hardness) results, and discuss NP-completeness and the P-versus-NP question. We also present the polynomial hierarchy and the boolean hierarchy over NP and list a number of problems complete in the higher levels of these two hierarchies.

1 Introduction

Complexity theory is an ongoing area of algorithm research that has demonstrated its practical value by steering us away from inferior algorithms. It also gives us an understanding about the level of inherent algorithmic difficulty of a problem, which affects how much effort we spend on developing sharp models that mitigate the computation time. It has also spawned approximation algorithms that, unlike metaheuristics, provide a bound on the quality of solution obtained in polynomial time.

This supplement is a brief introduction to the theory of computational complexity, which in particular provides important notions, techniques, and results to classify problems in terms of their complexity. For a more detailed and more comprehensive introduction to this field, we refer to the textbooks by Papadimitriou [Pap94], Rothe [Rot05], and Wilf [Wil94], and also to Tovey's tutorial [Tov02]. More specifically, Garey and Johnson's introduction to the theory

[*]This supplement replaces Harvey Greenberg's original supplement on complexity [Gre01]; it is based on and extends his text and uses parts of it with Harvey Greenberg's kind permission. This work was supported in part by the DFG under grants RO 1202/12-1 (within the European Science Foundation's EUROCORES program LogICCC: "Computational Foundations of Social Choice") and RO 1202/11-1.

of NP-completeness [GJ79] and Johnson's ongoing NP-completeness columns [Joh81] are very recommendable additional sources.

This paper is organized as follows. In Section 2, we briefly describe the foundations of complexity theory by giving a formal model of algorithms in terms of Turing machines and by defining the complexity measure time. We then survey upper bounds on the time complexity of selected problems and analyze Dijkstra's algorithm as an example. In Section 3, we are concerned with the two most central complexity classes, P and NP, deterministic and nondeterministic polynomial time. We define the notion of polynomial-time many-one reducibility, a useful tool to compare problems according to their complexity, and we discuss the related notions of NP-hardness and NP-completeness as well as the P-versus-NP question, one of the most fundamental challenges in theoretical computer science. In Section 4, further complexity classes and hierarchies between polynomial time and polynomial space are introduced in order to show a bit of complexity theory beyond P and NP. In particular, the polynomial hierarchy is discussed in Section 4.1 and the boolean hierarchy over NP in Section 4.2 We also list a number of problems complete in complexity classes above NP, namely, in the higher levels of the two hierarchies just mentioned. Section 5, finally, provides some concluding remarks.

2 Foundations of Complexity Theory

2.1 Algorithms, Turing Machines, and Complexity Measures

An *algorithm* is a finite set of rules or instructions that when executed on an input can produce an output after finitely many steps. The input encodes an instance of the problem that the algorithm is supposed to solve, and the output encodes a solution for the given problem instance. The *computation of an algorithm on an input* is the sequence of steps it performs. Of course, algorithms do not always terminate; they may perform an infinite computation for some, or any, input (consider, for example, the algorithm that ignores its input and enters an infinite loop). There are problems that can be easily and precisely defined in mathematical terms, yet that provably cannot be solved by any algorithm. Such problems are called *undecidable* (when they are decision problems, i.e., when they are described by a yes/no question) or *uncomputable* (when they are functions). Interestingly, the problem of whether a given algorithm ever terminates for a given input is itself algorithmically unsolvable, and perhaps is the most famous undecidable problem, known as the HALTING PROBLEM (see the seminal paper by Alan Turing [Tur36]). Such results—establishing a sharp boundary between algorithmically decidable and undecidable problems—are at the heart of computability theory, which forms the foundation of theoretical computer science. Computational complexity theory has its roots in computability theory, yet it takes things a bit further. In particular, we focus here on "well-behaved" problems that are algorithmically solvable, i.e., that can be solved by algorithms that on any input terminate after a finite number of steps have been executed. Note, however, that a computable function need not be total in general and, by definition, an algorithm computing it does never terminate on inputs on which the function is not defined.

In complexity theory, we are interested in the amount of resources (most typically, computation time or memory) needed to solve a given problem. To formally describe a problem's inherent complexity, we first need to specify

1. what tools we have for solving it (i.e., the computational model used) and
2. how they work (i.e., the computational paradigm used), and
3. what we mean by "complexity" (i.e., the complexity measure used).

First, the standard *computational model* is the Turing machine, and we give an informal definition and a simple concrete example below. Alternatively, one may choose one's favorite model among a variety of formal computation models, including the λ calculus, register machines, the notion of partial recursive function, and so on (see, e.g., [Rog67, HU79, Odi89, HMU01]). Practitioners may be relieved to hear that their favorite programming language can serve as a computational model as well. The famous Church–Turing thesis postulates that each of these models captures the notion of what *intuitively* is thought of being computable. This thesis (which by involving a vague intuitive notion necessarily defies a formal proof) is based on the observation that all these models are equivalent in the sense that each algorithm expressed in any of these models can be transformed into an algorithm formalized in any of the other models such that both algorithms solve the same problem. Moreover, the transformation itself can be done by an algorithm. One of the reasons the Turing machine has been chosen to be the standard model of computation is its simplicity.

Second, there is a variety of *computational paradigms* that yield diverse acceptance behaviors of the respective Turing machines, including deterministic, nondeterministic, probabilistic, alternating, unambiguous Turing machines, and so on. The most basic paradigms are determinism and nondeterminism, and the most fundamental complexity classes are P (deterministic polynomial time) and NP (nondeterministic polynomial time), which will be defined below.

Finally, as mentioned above, the most typical *complexity measures* are computation time (i.e., the number of steps a Turing machine performs for a given input) and *computation memory* (a.k.a. computation space, i.e., the number of tape cells a Turing machine uses during a computation as will be explained below).

Turing machines are simple, abstract models of algorithms. To give an informal description, imagine such a machine to be equipped with a finite number of (infinite) tapes each being subdivided into cells that are filled with a symbol from the work alphabet. To indicate that a cell is empty, we use a blank symbol, $\square$, which also belongs to the work alphabet. The machine accesses its tapes via heads, where each head scans one cell at a time. Heads can be read-only (for input tapes), they can be read-write (for work tapes), and they can be write-only (for output tapes). In addition, the machine uses a "finite control" unit as an internal memory; at any time during a computation the machine is in one of a finite number of states. One distinguished state is called the initial state, and there may be (accepting/rejecting) final states.[1] Finally, the actual program of the Turing machine consists of a finite set of rules that all have the following form (for each tape):

$$(z,a) \mapsto \left(z',b,H\right),$$

[1]For decision problems, whenever a Turing machine halts, it must indicate whether or not the input is to be accepted or rejected. Thus, in this case each final state is either an accepting or a rejecting state. In contrast, if a Turing machine is to compute a function then, whenever it halts in a final state (where the distinction between accepting and rejecting states is not necessary), the string written on the output tape is the function value computed for the given input.

where z and z' are states, a and b are symbols, and $H \in \{\leftarrow, \downarrow, \rightarrow\}$ is the move of the head: one cell to the left ($\leftarrow$), one cell to the right ($\rightarrow$), or staying at the same position ($\downarrow$). This rule can be applied if and only if the machine is currently in state z and this tape's head currently reads a cell containing the symbol a. To apply this rule then means to switch to the new state z', to replace the content of the cell being read by the new symbol b and to move this tape's head according to H.

Example 2.1 *For simplicity, in this example we consider a machine with only one tape, which initially holds the input string, symbol by symbol in consecutive tape cells, and which also serves as the work tape and the output tape. The Turing machine whose program is given in Table 1 takes a string $x \in \{0,1\}^*$ as input and computes its complementary string $\overline{x} \in \{0,1\}^*$, i.e., the i-th bit of $\overline{x}$ is a one if and only if the i-th bit of x is a zero (e.g., if $x = 10010$ then $\overline{x} = 01101$).*

$(z_0, 0)$	$\mapsto$	$(z_0, 1, \rightarrow)$	swap zeros to ones and step one cell to the right
$(z_0, 1)$	$\mapsto$	$(z_0, 0, \rightarrow)$	swap ones to zeros and step one cell to the right
$(z_0, \square)$	$\mapsto$	$(z_1, \square, \leftarrow)$	switch to z_1 at the end of the input and step left
$(z_1, 0)$	$\mapsto$	$(z_1, 0, \leftarrow)$	leave the letters unchanged and step one cell to the left
$(z_1, 1)$	$\mapsto$	$(z_1, 1, \leftarrow)$	until reaching the beginning of data on the tape
$(z_1, \square)$	$\mapsto$	$(z_F, \square, \rightarrow)$	move the head onto the first letter and switch to z_F to halt

Table 1: A Turing machine

The work alphabet is $\{\square, 0, 1\}$. A configuration *(a.k.a.* instantaneous description*) is some sort of "snapshot" of the work of a Turing machine in each (discrete) point in time. Each configuration is characterized by the current state, the current tape inscriptions, and the positions where the heads currently are. In particular, a Turing machine's initial configuration is given by the input string that is written on the input tape (while all other tapes are empty initially), the input head reading the leftmost symbol of the input (while the other heads are placed on an arbitrary blank cell of their tapes initially), and the initial state of the machine. One may also assume Turing machines to be normalized, which means they have to move their head(s) back to the leftmost nonblank position(s) before terminating. Our Turing machine in Table 1 has three states:*

- z_0*, which is the initial state and whose purpose it is to move the head from left to right over x swapping its bits, thus transforming it into* $\overline{x}$*,*
- z_1*, whose purpose it is to move the head back to the first bit of* $\overline{x}$ *(which was the head's initial position), and*
- z_F*, which is the final state.*

Unless stated otherwise, we henceforth think of an algorithm as being implemented by a Turing machine. A Turing machine is said to be *deterministic* if any two Turing rules in its program have distinct left-hand sides. This implies that in each computation of a deterministic Turing machine there is a unique successor configuration for each nonfinal configuration. Otherwise (i.e., if a Turing machine has two or more rules with identical left-hand sides), there can be two or more distinct successor configurations of a nonfinal configuration in some computation of this Turing machine, and in this case it is said to be *nondeterministic*.

The *computation of a deterministic Turing machine on some input* is a sequence of configurations, either a finite sequence that starts from the initial configuration and terminates with a final configuration (i.e., one with a final state) if one exists, or an infinite sequence that represents a nonterminating computation (e.g., a computation that is looping forever). In contrast, the *computation of a nondeterministic Turing machine on some input* is a directed tree whose root is the initial configuration, whose leaves are the final configurations, and where the rules of the Turing program describe one computation step from some nonfinal configuration to (one of) its successor(s).[2] There may be infinite and finite paths in such a tree for the same input. For the machine to terminate it is enough that just one computation path terminates. In the case of decision problems, for the input to be accepted it is enough that just one computation path leads to a leaf with an accepting final configuration.

The *time complexity of a deterministic Turing machine M on some input x* is the number $\mathit{time}_M(x)$ of steps $M(x)$ needs to terminate. If M on input x does not terminate, $\mathit{time}_M(x)$ is undefined; this is reasonable if one takes the point of view that any computation should be assigned a complexity if and only if it yields a result, i.e., if and only if it terminates. The *space complexity of M on input x* is defined analogously by the number of nonblank tape cells needed until termination. In this paper, however, we will focus on the complexity measure time only (see, e.g., [Pap94], Rothe [Rot05] for the definition of space complexity).

The most common model is to consider the *worst-case complexity* of a Turing machine M as a function of the input size $n = |x|$,[3] i.e., we define $\mathit{Time}_M(n) = \max_{x:|x|=n} \mathit{time}_M(x)$ if $\mathit{time}_M(x)$ is defined for all x of length n, and we let $\mathit{Time}_M(n)$ be undefined otherwise. (We omit a discussion of the notion of *average-case complexity* here and refer to the work of Levin [Lev86] and Goldreich [Gol97].)

Now, given a time complexity function $t : \mathbb{N} \rightarrow \mathbb{N}$, we say a (possibly partial) function f is *deterministically computable in time t* if there exists a deterministic Turing machine M such that for each input x,

(a) $M(x)$ terminates if and only if $f(x)$ is defined,

(b) if $M(x)$ terminates then $M(x)$ outputs $f(x)$, and

(c) $\mathit{Time}_M(n) \leq t(n)$ for all but finitely many $n \geq 0$.

(There are various notions of what it means for a *nondeterministic* Turing machine to compute a function in a given time. We omit this discussion here and refer to the work of Selman [Sel94].)

[2]Note that a deterministic computation can be considered as a (degenerated) computation tree as well, namely a tree that degenerates to a single path. That is, deterministic algorithms are special nondeterministic algorithms.

[3] We always assume a reasonable, natural encoding of the input over some input alphabet such as $\{0,1\}$, and n denotes the length of the input $x \in \{0,1\}^*$ in this encoding. However, since one usually can neglect the small change of time complexity caused by the encoding used, for simplicity n is often assumed to be the "typical" parameter of the input. This may be the number of digits if the input is a number, or the number of entries if the input is a vector, or the number of vertices if the input is a graph, or the number of variables if the input is a boolean formula, etc.

We now define the time complexity of decision problems. Such a problem is usually represented either by stating the input given and a related yes/no question, or as a set of strings encoding the "yes" instances of the problem. For example, the satisfiability problem of propositional logic, SAT, asks whether a given boolean formula is satisfiable, i.e., whether there is an assignment of truth values to its variables such that the formula evaluates to true. Represented as a set of strings (i.e., as a "language" over some suitable alphabet) this problem takes the form:

$$\mathrm{SAT} = \{\varphi \mid \varphi \text{ is a satisfiable boolean formula}\}. \tag{1}$$

Given a time complexity function $t : \mathbb{N} \to \mathbb{N}$, a decision problem D is solved (or "decided") by a deterministic Turing machine M in time t if

(a) M accepts exactly the strings in D and

(b) $\mathit{Time}_M(n) \leq t(n)$ for all but finitely many $n \geq 0$.

We say D is solved (or "accepted") by a nondeterministic Turing machine M in time t if

(a) for all $x \in D$, the computation tree $M(x)$ has at least one accepting path and the shortest accepting path of $M(x)$ has length at most $t(|x|)$, and

(b) for all $x \notin D$, the computation tree $M(x)$ has no accepting path.

Let $\mathrm{DTIME}(t)$ denote the class of problems solvable by deterministic Turing machines in time t, and let $\mathrm{NTIME}(t)$ denote the class of problems solvable by nondeterministic Turing machines in time t.

The primary focus of complexity theory is on the computational complexity of decision problems. However, mathematicians are often more interested in computing (numerical) solutions of functions. Fortunately, the complexity of a functional problem is usually closely related to that of the corresponding decision problem. For example, to decide whether a linear program (LP, for short) has a feasible solution is roughly as hard as finding one.

2.2 Upper Bounds on the Time Complexity of Some Problems

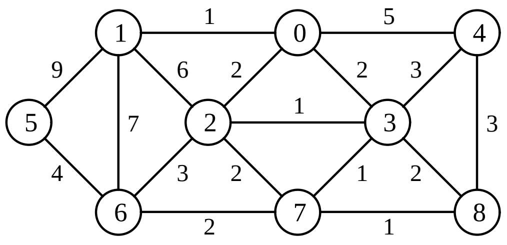
Is a problem with time complexity $17 \cdot n^2$ more difficult to solve than a problem with time complexity n^2? Of course not. Constant factors (and hence also additive constants) can safely be ignored. Already the celebrated papers by Hartmanis, Lewis, and Stearns [HS65, SHL65, HLS65, LSH65] (which laid the foundations of complexity theory and for which Hartmanis and Stearns won the Turing Award) proved the linear speed-up theorem: For each time complexity function t that grows strictly stronger than the identity and for each constant c, $\mathrm{DTIME}(t) = \mathrm{DTIME}(c \cdot t)$. An analogous result is known for the nondeterministic time classes $\mathrm{NTIME}(t)$ (and also for deterministic and nondeterministic space classes, except there one speaks of linear compression rather than of linear speed-up.)

This result gives rise to define the "big O" notation to collect entire families of time complexity functions capturing the algorithmic upper bounds of problems. Let $f, g : \mathbb{N} \longrightarrow \mathbb{N}$ be two functions. We say *f grows asymptotically no stronger than g* (denoted by $f \in \mathscr{O}(g)$) if there exists a real number $c > 0$ and a nonnegative integer n_0 such that for each $n \in \mathbb{N}$ with $n \geq n_0$, we have $f(n) \leq c \cdot g(n)$. Alternatively, $f \in \mathscr{O}(g)$ if and only if there is a real number $c > 0$ such that $\limsup_{n\to\infty} {}^{(f(n)+1)}\!/\!_{(g(n)+1)} = c$. In particular, DTIME$(\mathscr{O}(n))$ denotes the class of problems solvable by deterministic linear-time algorithms. Table 2 lists some problems and some upper bounds on their deterministic time complexity.

DTIME$(\cdot)$	time complexity	sample problem
$\mathscr{O}(1)$	constant	determine the signum of a number
$\mathscr{O}(\log(n))$	logarithmic	searching in a balanced search tree with n nodes
$\mathscr{O}(n)$	linear	solving differential equations using multigrid methods
$\mathscr{O}(n \cdot \log(n))$		sort n numbers (mergesort)
$\mathscr{O}(n^2)$	quadratic	compute the product of an $n \times n$ matrix and a vector
$\mathscr{O}(n^3)$	cubic	Gaussian elimination for an $n \times n$ matrix
$\mathscr{O}(n^k)$, k fixed	polynomial	solving LPs using interior point methods
$\mathscr{O}(k^n)$, k fixed	exponential	solving LPs using the simplex method

Table 2: Upper bounds on the worst-case time complexity of some problems

If there are distinct algorithms for solving the same problem, we are usually interested in the one with the best worst-case behavior, i.e., we would like to know the least upper bound of the problem's worst-case complexity. Note that the actual time required for a particular instance could be much less than this worst-case upper bound. For example, Gaussian elimination applied to a linear system of equations, $Ax = b$ with $A \in \mathbb{Q}^{n\times n}$ and a vector $b \in \mathbb{Q}^n$,[4] has a worst-case time complexity of $\mathscr{O}(n^3)$, but would be solved much faster if A happens to be the identity matrix. However, the *worst-case* complexity relates to the most stubborn inputs of a given length.


(a) Graph $G = (V, E)$

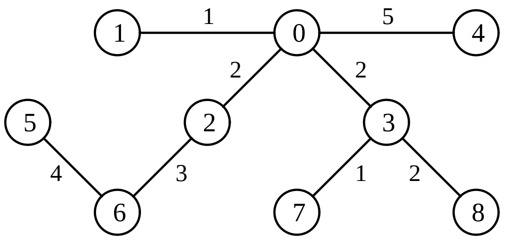

(b) Graph $H = (V, F)$ containing the shortest paths from G with start vertex $s = 0$

Figure 1: Applying Dijkstra's algorithm to graph G yields graph H

For a concrete example, consider the shortest path problem, which is defined as follows: Given a network (i.e., an undirected graph $G = (V, E)$ with positive edge weights given by $f : E \to \mathbb{Q}_{\geq 0}$)

[4]We assume A to be an $n\times n$ matrix over the rational numbers, $\mathbb{Q}$, and x and b to be vectors over $\mathbb{Q}$, since complexity theory traditionally is concerned with discrete problems. A complexity theory of real numbers and functions has also been developed. However, just as in numerical analysis, this theory actually approximates irrational numbers up to a given accuracy.

and a designated vertex $s \in V$, find all shortest paths from s to each of the other vertices in V. For illustration, we show how an upper bound on the worst-case time complexity of this problem can be derived. Let $V = \{0,1,2,3,4,5,6,7,8\}$ and the edge set E be as shown in Figure 1(a). Let $f : E \to \mathbb{Q}_{\geq 0}$ be the edge weight function, and let $s = 0$ be the designated start vertex. Dijkstra's algorithm (see Algorithm 1 for a pseudocode description; a Turing machine implementation would be rather cumbersome) finds a shortest path from vertex $s = 0$ to each of the other vertices in G.

Algorithm 1 Dijkstra's algorithm

```
1:  input: $(G = (V,E), f)$ with start vertex $s = 0$
2:  $n := |V| - 1$                                   // set $n$ to the highest vertex number
3:  for $v = 1,2,\ldots,n$ do
4:      $\ell(v) := \infty$;                         // $\ell(v)$ is the length of the shortest path from $s = 0$ to $v$
5:      $p(v) :=$ undefined                          // $p(v)$ is the predecessor of $v$ on the shortest path from $s$ to $v$
6:  end for
7:  $\ell(0) := 0$;
8:  $W := \{0\}$                                     // insert the start vertex into $W$, the set of vertices still to check
9:  while $W \neq \emptyset$ do
10:     Find a vertex $v \in W$ such that $\ell(v)$ is minimum;
11:     $W := W - \{v\}$;                            // extract $v$ from $W$
12:     for all $\{v,w\} \in E$ do
13:         if $\ell(v) + f(\{v,w\}) < \ell(w)$ then
14:             $\ell(w) := \ell(v) + f(\{v,w\})$    // update the length of the shortest path to $w$ if needed
15:             $p(v) := w$                          // update the predecessor of $w$ if needed
16:             if $w \notin W$ then
17:                 $W := W \cup \{w\}$              // insert neighbors of $v$ into $W$ (if not yet done)
18:             end if
19:         end if
20:     end for
21: end while
22: $F := \emptyset$                                 // $F$ will be the edge set containing the shortest paths
23: for $v = 1,2,\ldots,n$ do
24:     $F := F \cup \{\{v, p(v)\}\}$                // insert the edge $\{v, p(v)\}$ into $F$
25: end for
26: output: $H = (V,F)$, the graph containing the shortest paths from $s = 0$
```

Table 3 gives an example for applying Disjkstra's algorithm to the input graph shown in Figure 1(a). The table shows the initialization and the execution of the **while** loop. The resulting graph $H = (V,F)$—with new edge set F—containing the shortest paths is shown in Figure 1(b).

A first rough estimate of the time complexity shows that Dijkstra's algorithm runs in time $\mathcal{O}(n^2)$, where $n = |V|$ is the number of vertices in G. The initializiation takes $\mathcal{O}(n)$ time, and so does the computation of F at the end of the algorithm. The **while** loop is executed n times. Since we choose a vertex v such that the length of a shortest path from s to v is minimum, each vertex will be added exactly once to W and thus can be deleted only once from W. Per execution of the **while**

Input: Graph $G=(V,E)$, $f: V \times V \to \mathbb{Q}_{\geq 0}$ (see Figure 1(a))																	
		$\ell(\cdot)$; note that $\ell(0)=0$								$p(\cdot)$; note that $p(0)$ doesn't exist							
v	W	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
-	$\{0\}$	∞	∞	∞	∞	∞	∞	∞	∞								
0	$\{1,2,3,4\}$	1	2	2	5	∞	∞	∞	∞	0	0	0	0				
1	$\{2,3,4,5,6\}$	1	2	2	5	10	8	∞	∞	0	0	0	0	1	1		
2	$\{3,4,5,6,7\}$	1	2	2	5	10	5	4	∞	0	0	0	0	1	2	2	
3	$\{4,5,6,7,8\}$	1	2	2	5	10	5	3	4	0	0	0	0	1	2	3	3
7	$\{4,5,6,8\}$	1	2	2	5	10	5	3	4	0	0	0	0	1	2	3	3
8	$\{4,5,6\}$	1	2	2	5	10	5	3	4	0	0	0	0	1	2	3	3
4	$\{5,6\}$	1	2	2	5	10	5	3	4	0	0	0	0	1	2	3	3
6	$\{5\}$	1	2	2	5	9	5	3	4	0	0	0	0	6	2	3	3
5	$\emptyset$	1	2	2	5	9	5	3	4	0	0	0	0	6	2	3	3

Table 3: Example of Dijkstra's algorithm applied to the graph from Figure 1(a)

loop the most expensive tasks are finding a vertex $v \in W$ such that $\ell(v)$ is minimum (which takes time $\mathcal{O}(n)$) and visiting v's neighbors (which again takes time $\mathcal{O}(n)$ in the worst case). If we use a priority queue (concretely implemented by a Fibonacci heap) for managing W, an amortized runtime analysis yields an upper bound of $\mathcal{O}(|V|\log|V|) + \mathcal{O}(|E|\log|V|) = \mathcal{O}((|V|+|E|)\log|V|)$, which is better than $\mathcal{O}(|V|^2)$ if the graph is sparse in edges.

3 Deterministic and Nondeterministic Polynomial Time

3.1 P, NP, Reducibility, and NP-Completeness

Complexity classes collect the problems that can be solved, according to the given computational model (such as Turing machines) and paradigm (e.g., determinism or nondeterminism), by algorithms of that sort using no more than the specified amount of the complexity resource considered (e.g., computation time or space). The most prominent time complexity classes are P and NP.

By definition, $\mathrm{P} = \bigcup_{k \geq 0} \mathrm{DTIME}(n^k)$ is the class of all problems that can be solved by a deterministic Turing machine in polynomial time, where the polynomial time bound is a function of the input size. For example, Dijkstra's algorithm (see Algorithm 1 in Section 2) is a polynomial-time algorithm solving the shortest path problem, which is a functional problem.[5] A decision version of that problem would be: Given a network $G=(V,E)$ with edge weights $f: E \to \mathbb{Q}_{\geq 0}$, a designated vertex $s \in V$, and a positive constant k, is the length of a shortest path from s to any of the other vertices in G bounded above by k? This problem is a member of P.

Nondeterministic polynomial time, defined as $\mathrm{NP} = \bigcup_{k \geq 0} \mathrm{NTIME}(n^k)$, is the class of (decision)

[5]As mentioned previously, complexity theory traditionally is concerned more with decision than with functional problems, so P and NP are collections of decision problems. The class of functions computable in polynomial time is usually denoted by FP, and there are various alternative ways of defining classes of functions computable in nondeterministic polynomial time (see [Sel94]).

problems for which solutions to the given input instance can be guessed and verified in polynomial time. In other words, problems in NP can be solved by a nondeterministic Turing machine in polynomial time. For example, consider the problem of whether a given system of diophantine equations, $Ax = b$ for a matrix $A \in \mathbb{Z}^{n\times n}$ and a vector $b \in \mathbb{Z}^n$, has an integer solution $x \in \mathbb{Z}^n$. We do not know whether there exists a polynomial-time algorithm to solve this problem, but for any given instance (A,b) of the problem, we can guess a solution, $x \in \mathbb{Z}^n$, and then verify in polynomial time that x indeed is a correct solution (simply by checking whether $A \in \mathbb{Z}^{n\times n}$ multiplied by $x \in \mathbb{Z}^n$ yields b, i.e., by checking whether x satisfies $Ax = b$). Thus, this problem is a member of NP. Another well-known member of NP is SAT, the satisfiability problem of propositional logic defined in (1). There are thousands of other important problems that are known to belong to NP (see, e.g., Garey and Johnson [GJ79]).

Having established a problem's upper bound is only the first step for determining its computational complexity. Only if we can establish a lower bound matching this upper bound, we will have classified our problem in terms of its complexity. But lower bound proofs are different. In order to prove an upper bound, it is enough to find just one suitable algorithm that solves this problem within the prespecified amount of time. For a lower bound, however, we need to show that among all algorithms of the given type (e.g., among all algorithms that can be realized via a nondeterministic Turing machine) no algorithm whatsoever can solve our problem in (asymptotically) less than the prespecified amount of time.

A key concept for proving lower bounds is that of reducibility. This concept allows us to compare the complexity of two problems: If we can reduce a problem A to a problem B then A is at most as hard as B. This is a fundamental property in complexity theory.

Definition 3.1 *Let A and B be two problems, both encoded over an alphabet Σ (i.e., $A, B \subseteq \Sigma^*$, where Σ^* denotes the set of strings over Σ). We say A* (polynomial-time many-one) reduces to *B (in symbols, $A \leq_{\mathrm{m}}^{\mathrm{p}} B$) if there is a total function $f : \Sigma^* \to \Sigma^*$ computable in polynomial time such that for each input string $x \in \Sigma^*$, $x \in A$ if and only if $f(x) \in B$.*

B is said to be NP-hard *if $A \leq_{\mathrm{m}}^{\mathrm{p}} B$ holds for all $A \in$* NP*, and B is said to be* NP-complete *if B is* NP-*hard and in* NP.

Note that NP-hard problems are not necessarily members of NP; they can be even harder than the hardest NP problems, i.e., harder than any NP-complete problem.

3.2 Some Examples of NP-Complete Problems

After the groundbreaking work of Hartmanis and Stearns in the early 1960s (see, e.g., [HS65]), a landmark followed in 1971 when Cook showed that the satisfiability problem, SAT, is NP-complete. This result was found independently by Levin [Lev73].

Theorem 1 (Cook's Theorem [Coo71]) SAT *is* NP-*complete.*

Intuitively, the proof of Theorem 1 starts from an arbitrary NP machine M (in a suitable encoding) and an arbitrary input x and transforms the pair (M,x) into a boolean formula $\varphi_{M,x}$ such that M on input x has at least one accepting computation path if and only if $\varphi_{M,x}$ is satisfiable. Since this transformation can be done in polynomial time, it is a $\leq_{\mathrm{m}}^{\mathrm{p}}$-reduction from an arbitrary NP set to SAT. Note that the formula $\varphi_{M,x}$ Cook constructed is in conjunctive normal form (i.e., $\varphi_{M,x}$ is a conjunction of disjunctions of literals); thus, this reduction even shows NP-hardness of CNF-SAT, the restriction of SAT to boolean formulas in conjunctive normal form.

The next milestone in complexity theory is due to Karp [Kar72], who showed NP-completeness of many combinatorial optimization problems from graph theory and other fields. Since then the list of problems known to be NP-complete has grown enormously (see, e.g., [GJ79, Joh81, Pap94, Rot05]). Note that completeness is a central notion for other complexity classes as well; for example, there are natural complete problems in classes such as nondeterministic logarithmic space (NL, for short), polynomial time (i.e., P), and polynomial space (PSPACE, for short). Note also that for classes such as NL and P, the polynomial-time many-one reducibility is too coarse. Indeed, it is known that *every* nontrivial set (i.e., every nonempty set other than Σ^*) in any of these classes is complete for the class under $\leq_{\mathrm{m}}^{\mathrm{p}}$-reductions. Therefore, one needs a finer reducibility (such as the logarithmic-space many-one reducibility) so as to define a meaningful notion of completeness in such classes.

Returning to the notion of NP-completeness, we have from Definition 3.1 and the transitivity of $\leq_{\mathrm{m}}^{\mathrm{p}}$ that a problem B is NP-complete if

1. $B \in \mathrm{NP}$, and
2. $A \leq_{\mathrm{m}}^{\mathrm{p}} B$ for some NP-complete problem A.

We give two examples of reductions showing NP-completeness of two more problems.

Example 3.1 *Define* 3-SAT *to be the set of all boolean formulas in conjunctive normal form with at most three literals per clause. That* 3-SAT *belongs to* NP *follows immediately from the fact that the more general problem* SAT *is in* NP.

To show that 3-SAT *is* NP*-hard, we show* CNF-SAT $\leq_{\mathrm{m}}^{\mathrm{p}}$ 3-SAT. *Let φ be a given boolean formula in conjunctive normal form. We transform φ into an equivalent formula ψ with no more than three literals per clause. Clauses of φ with one, two, or three literals can be left untouched; we just have to take care of φ's clauses with more than three literals. Let $C = (z_1 \vee z_2 \vee \cdots \vee z_k)$ be one such clause with $k \geq 4$ literals (so each z_i is either a variable of φ or its negation). Let $y_1, y_2, \ldots, y_{k-3}$ be $k-3$ new variables. Define a formula ψ_C consisting of the following $k-2$ clauses:*

$$\psi_C \quad = \quad (z_1 \vee z_2 \vee y_1) \wedge (\neg y_1 \vee z_3 \vee y_2) \wedge \ldots \wedge (\neg y_{k-4} \vee z_{k-2} \vee y_{k-3}) \wedge (\neg y_{k-3} \vee z_{k-1} \vee z_k). \quad (2)$$

Note that C is satisfiable if and only if the formula ψ_C in (2) is satisfiable. Hence, replacing all clauses C in φ with more than three literals by the corresponding subformula ψ_C, we obtain a formula ψ that is satisfiable if and only if φ is satisfiable. Clearly, the reduction can be done in polynomial time. Thus 3-SAT *is* NP*-hard, and therefore also* NP*-complete.*

Example 3.2 *Consider the following problem: Given a $k \times n$ matrix A and an $m \times n$ matrix B with entries $A_{ij}, B_{ij} \in \{0,1\}$, does there exist a vector $x \in \{0,1\}^n$ such that $Ax = 1$ and $Bx \geq 1$?*

This problem is in NP *since any solution guessed, $x \in \{0,1\}^n$, can be certified by a polynomial-time algorithm that multiplies both A and B by x and tests if $Ax = 1$ and $Bx \geq 1$. We shall show that*

SAT $\leq_m^p$*-reduces to this problem. Suppose we are given a boolean formula* $\varphi = C_1 \wedge \cdots \wedge C_m$ *with propositions* $P_1, \ldots, P_k$ *and clauses* $C_1, \ldots, C_m$*. Define* $n = 2k$ *variables,* $x_{-1}, \ldots, x_{-k}, x_1, \ldots, x_k$*. Let* $A = [\mathrm{I}\;\mathrm{I}]$*, where* I *denotes the* $k \times k$ *identity matrix; so the system of equations,* $Ax = 1$*, corresponds to defining the logical complements:* $x_{-j} + x_j = 1$ *for all* j*. (Think of* $x_j = 1 \Leftrightarrow P_j =$ TRUE*, and* $x_{-j} = 1 \Leftrightarrow \neg P_j =$ TRUE*.)*

Now define the $m \times n$ *matrix* B *by*

$$B_{ij} = \begin{cases} 1 & \text{if } P_j \in C_i \\ 0 & \text{otherwise} \end{cases} \quad \text{and} \quad B_{i,-j} = \begin{cases} 1 & \text{if } \neg P_j \in C_i \\ 0 & \text{otherwise.} \end{cases}$$

Any solution to this system corresponds to a satisfying assignment for φ *by letting* $P_j =$ TRUE *if and only if* $x_j = 1$.

More concretely, given a formular $\varphi = (P_1 \vee P_3) \wedge (P_2 \vee \neg P_3) \wedge (\neg P_1 \vee \neg P_2 \vee P_3)$*, the transformation creates the following system of equations and inequalities:*

$$\begin{array}{ccccccccccccc}
x_{-1} & & & & & + & x_1 & & & & & = & 1 \\
& & x_{-2} & & & & & + & x_2 & & & = & 1 \\
& & & & x_{-3} & & & & & + & x_3 & = & 1 \\
\\
& & & & & & x_1 & & & + & x_3 & \geq & 1 \\
& & & & x_{-3} & & & + & x_2 & & & \geq & 1 \\
x_{-1} & + & x_{-2} & & & & & & & + & x_3 & \geq & 1
\end{array}$$

The three equations define the logical complements, $x_{-j} = 1 - x_j$ *for* $j \in \{1,2,3\}$*. The three inequalities ensure each clause has at least one literal that is true. In this example, a solution to the above system is* $x = (0,0,0,1,1,1)$*; the corresponding truth assignment satisfying* φ *is* $P_j =$ TRUE *for all* $j \in \{1,2,3\}$.

Since every deterministic Turing machine is a special case of a nondeterministic one, P is a subclass of NP. Whether or not these two classes are equal is a famous open question, perhaps the most important open question in all of theoretical computer science. If one could find a deterministic polynomial-time algorithm for solving an arbitrary NP-complete problem, then P would be equal to NP. However, it is widely believed that P $\neq$ NP.

Returning to LPs, how hard is linear programming? That is, how hard is it to solve

$$\min\{c^T \cdot x \mid A \cdot x \geq b\}$$

for $x \in \mathbb{Q}^n$, where $c \in \mathbb{Q}^n$, $A \in \mathbb{Q}^{m \times n}$, and $b \in \mathbb{Q}^m$ are given? Although every known simplex method has an exponential worst-case time complexity, some interior point methods are polynomial. Thus, the decision version of linear programming is a member of P. However, if we are looking for an integer solution $x \in \mathbb{Z}^n$, the problem is known to be NP-complete.[6]

Despite their NP-completeness, problems that (unlike those defined in Examples 3.1 and 3.2) involve numbers that can grow unboundedly may have “pseudo-polynomial-time” algorithms.

[6] In order to use the interior point methods, which have a polynomial runtime, it is necessary to transform the problem; this causes an exponential number of additional constraints, so the problem size increases exponentially.

Informally stated, an algorithm solving some problem involving numbers is said to be *pseudo-polynomial-time* if its time complexity function is bounded above by a polynomial function of two variables, the size of the problem instance and the maximum value of the numbers involved. For a formal definition, we refer to Garey and Johnson [GJ79]. For example, consider the problem PARTITION: Given a nonempty sequence $a_1, a_2, \ldots, a_m$ of positive integers such that $\sum_{i=1}^{m} a_i$ is an even number, can one find a partition $I_1 \cup I_2 = \{1, 2, \ldots, m\}$, $I_1 \cap I_2 = \emptyset$, such that $\sum_{i \in I_1} a_i = \sum_{i \in I_2} a_i$? This problem is known to be NP-complete but can be solved in pseudo-polynomial time via dynamic programming. The same is true for the knapsack problem[7] and many more NP-complete problems involving numbers. That is why Garey and Johnson [GJ79] distinguish between "ordinary" NP-completeness as defined in Definition 3.1 and NP-completeness "in the strong sense."

An NP-complete problem is NP-*complete in the strong sense* if it cannot be solved even in pseudo-polynomial time unless P = NP. As mentioned above, PARTITION is an example of a problem that is NP-complete but not in the strong sense. For an example of a problem that *is* NP-complete in the strong sense, consider the following generalization of PARTITION (which is called 3-PARTITION in [GJ79]): Given a positive integer b and a set $A = \{a_1, a_2, \ldots, a_{3m}\}$ of elements with positive integer weights $w(a_i)$ satisfying $b/4 < w(a_i) < b/2$ for each i, $1 \leq i \leq 3m$, such that $\sum_{i=1}^{3m} w(a_i) = m \cdot b$, can one partition A into m disjoint subsets, $A_1 \cup A_2 \cup \cdots \cup A_m = A$, such that for each j, $1 \leq j \leq m$, $\sum_{a_i \in A_j} w(a_i) = b$? In contrast, SAT involves no numbers except as subscripts (which can be ignored,[8] as they only refer to the names of variables or literals); for such problems there is no difference between polynomial time and pseudo-polynomial time and the notion of NP-completeness in the strong sense coincides with ordinary NP-completeness.

The complexity class $\mathrm{coNP} = \{L \mid \overline{L} \in \mathrm{NP}\}$ contains those problems whose complements are in NP. An example of a member of coNP is the tautology problem: Given a boolean formula, is it a tautology, i.e., is it true under every possible truth assignment to its variables? Note that coNP expresses the power of (polynomially length-bounded) universal quantification (i.e., is some polynomial-time predicate $B(x,y)$ true *for all* solutions y, where $|y| \leq p(|x|)$ for some polynomial p?), whereas NP expresses the power of (polynomially length-bounded) existential quantification (i.e., is some polynomial-time predicate $B(x,y)$ true *for some* solution y, where $|y| \leq p(|x|)$ for some polynomial p?).

Again, it is an open question whether or not NP and coNP are equal. Since P is closed under complementation, P = NP implies NP = coNP; the converse, however, is not known to hold. The notions of hardness and completeness straightforwardly carry over to coNP; for example, the tautology problem is known to be coNP-complete. Figure 2 illustrates the complexity classes discussed in this section and what is currently known about their inclusion relations.

[7] Glover and Babayev [GB95] show how to aggregate integer-valued diophantine equations whose variables are restricted to the nonnegative integers such that the coefficients of the new system of equations are in a range as limited as possible. Their methods can be applied to yield more efficient algorithms for the integer knapsack problem. For a comprehensive treatise of the famous knapsack problem and its variants, we refer to Kellerer et al. [KPP04].

[8] Technically speaking, as mentioned in Footnote 3, our assumptions on encoding imply that such "numerical" names are always polynomially bounded in the input size.

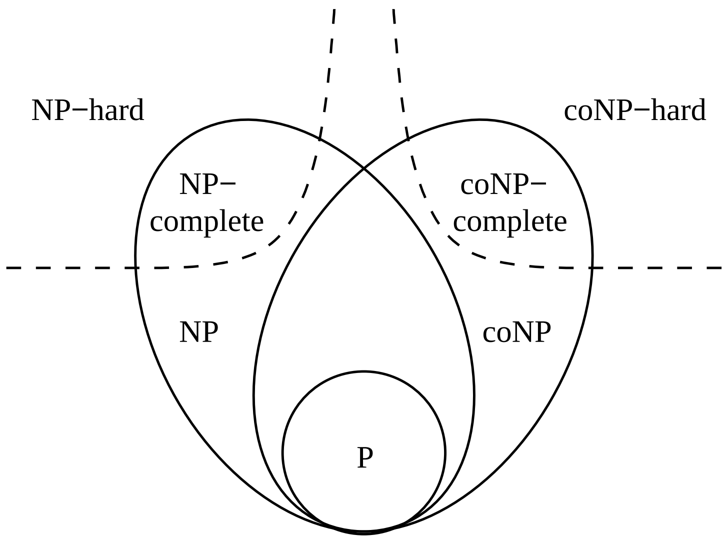


Figure 2: P, NP, coNP, and completeness and hardness for NP and coNP

4 Complexity Classes and Hierarchies between P and PSPACE

As mentioned previously, the most important complexity classes are P and NP. However, the landscape of complexity classes is much richer. In what follows, we mention some of the complexity classes and hierarchies between P and PSPACE that have been studied.

4.1 The Polynomial Hierarchy

Meyer and Stockmeyer [MS72] introduced the *polynomial hierarchy,* which is inductively defined by:

$$\begin{aligned}
\Delta_0^p &= \Sigma_0^p = \Pi_0^p = \mathrm{P};\\
\Delta_{i+1}^p &= \mathrm{P}^{\Sigma_i^p}, \quad \Sigma_{i+1}^p = \mathrm{NP}^{\Sigma_i^p}, \text{ and } \quad \Pi_{i+1}^p = \mathrm{co}\Sigma_{i+1}^p \quad \text{for } i \geq 0;\\
\mathrm{PH} &= \bigcup_{k\geq 0} \Sigma_k^p.
\end{aligned}$$

Here, for any two complexity classes $\mathscr{C}$ and $\mathscr{D}$, $\mathscr{C}^{\mathscr{D}}$ is the class of sets that can be accepted via some $\mathscr{C}$ oracle Turing machine that accesses an oracle set $D \in \mathscr{D}$. Oracle Turing machines work just like ordinary Turing machines, except they are equipped with a query tape on which they can write query strings. Whenever the machine enters a special state, the query state, it receives the answer as to whether the string currently written on the query tape, say q, belongs to the oracle set D or not. The machine continues its computation in the “yes” state if $q \in D$, and in the “no” state otherwise. As a special case, P^{NP} contains all problems that are polynomial-time Turing-reducible to some set in NP, i.e., $A \in \mathrm{P}^{\mathrm{NP}}$ if and only if there exists a deterministic polynomial-time oracle Turing machine M and an oracle set $B \in \mathrm{NP}$ such that M^B accepts precisely the strings in A via queries to B.

The first three levels of the polynomial hierarchy consist of the following classes:

$$\begin{array}{lcllcllcl}
\Delta_0^p & = & \mathrm{P}, & \Sigma_0^p & = & \mathrm{P}, & \Pi_0^p & = & \mathrm{P}; \\
\Delta_1^p & = & \mathrm{P}^{\mathrm{P}} = \mathrm{P}, & \Sigma_1^p & = & \mathrm{NP}^{\mathrm{P}} = \mathrm{NP}, & \Pi_1^p & = & \mathrm{coNP}^{\mathrm{P}} = \mathrm{coNP}; \\
\Delta_2^p & = & \mathrm{P}^{\mathrm{NP}}, & \Sigma_2^p & = & \mathrm{NP}^{\mathrm{NP}}, & \Pi_2^p & = & \mathrm{coNP}^{\mathrm{NP}}.
\end{array}$$

Let us summarize the known inclusions among these classes and between PH and PSPACE and some of their basic properties:

1. For each $i \geq 0$, $\Sigma_i^p \cup \Pi_i^p \subseteq \Delta_{i+1}^p \subseteq \Sigma_{i+1}^p \cap \Pi_{i+1}^p$.

2. PH $\subseteq$ PSPACE.

3. Each of the classes Δ_i^p, Σ_i^p, Π_i^p, $i \geq 0$, and PH is closed under $\leq_{\mathrm{m}}^{\mathrm{p}}$-reductions (e.g., if $A \leq_{\mathrm{m}}^{\mathrm{p}} B$ and $B \in \Sigma_2^p$ then $A \in \Sigma_2^p$).

4. The Δ_i^p levels of the polynomial hierarchy are closed even under polynomial-time Turing reductions (e.g., $\mathrm{P}^{\mathrm{P}^{\mathrm{NP}}} \subseteq \mathrm{P}^{\mathrm{NP}}$ shows this property for Δ_2^p).

5. Each of the classes Δ_i^p, Σ_i^p, and Π_i^p, $i \geq 0$, has $\leq_{\mathrm{m}}^{\mathrm{p}}$-complete problems. However, if PH were to have a $\leq_{\mathrm{m}}^{\mathrm{p}}$-complete problem then it would collapse to some finite level.

6. If $\Sigma_i^p = \Pi_i^p$ for some $i \geq 1$, then PH collapses to its i-th level:

$$\Sigma_i^p = \Pi_i^p = \Sigma_{i+1}^p = \Pi_{i+1}^p = \cdots = \mathrm{PH}.$$

Again, it is unknown whether any of the classes in the polynomial hierarchy are actually different. It is widely believed, however, that the polynomial hierarchy has infinitely many distinct levels; so a collapse to some finite level is considered unlikely. Note that the important P-versus-NP question mentioned previously is a special case of the question of whether the polynomial hierarchy is a strictly infinite hierachy.

The Σ_i^p levels of the polynomial hierarchy capture the power of i (polynomially length-bounded) alternating quantifers starting with an existential quantifer. For example, a typical Σ_3^p predicate is to ask whether there exists some y_1 such that for all y_2 there exists some y_3 such that some polynomial-time predicate $B(x,y_1,y_2,y_3)$ is true, where $|y_j| \leq p(|x|)$ for each $j \in \{1,2,3\}$. To give a concrete example, a chess player may face a question of this type: "Does there exist a move for White such that whatever move Black does there is a winning move for White?" In suitable formalizations of strategy games like chess (on an $n \times n$ board) this gives rise to a Σ_3^p-complete problem. Similarly, the Π_i^p levels of the polynomial hierarchy capture the power of i (polynomially length-bounded) alternating quantifers starting with a universal quantifer; see, e.g., Wrathall [Wra77]. When there is no bound on the number of alternating quantifiers (or moves), one obtains complete problems for PSPACE in this manner.

There are also interesting subclasses of Δ_i^p named Θ_i^p (see, e.g., Wagner [Wag87]). The most important among these classes is $\Theta_2^p = \mathrm{P}_{||}^{\mathrm{NP}} = \mathrm{P}^{\mathrm{NP}[\log]}$, which is defined just like P^{NP} except that *only a logarithmic number of queries* to the NP oracle are allowed, as indicated by the notation $\mathrm{P}^{\mathrm{NP}[\log]}$.

Alternatively, one could define Θ_2^p just like P^{NP} except that all queries to the NP oracle are required to be asked *in parallel*, as indicated by the notation $\mathrm{P}_{||}^{\mathrm{NP}}$, i.e., queries to the oracle do not depend on the oracle answers to previously asked queries, a less flexible query mechanism than in a Turing reduction. To give an example of a quite natural complete problem in this class, Hemaspaandra et al. [HHR97] proved that the winner determination problem for Dodgson elections is Θ_2^p-complete.

We give some more examples for problems contained (or even complete) in Σ_2^p, Π_2^p, Δ_2^p, and Θ_2^p, respectively:

- Meyer and Stockmeyer [MS72] introduced the problem MINIMAL: Given a boolean formula φ, does it hold that there is no shorter formula equivalent to φ? It is not hard to see that MINIMAL is in Π_2^p. Indeed, motivated by this problem they created the polynomial hierarchy.

- Garey and Johnson [GJ79] defined a variant of MINIMAL, which they dubbed MINIMUM EQUIVALENT EXPRESSION (MEE): Given a boolean formula φ and a nonnegative integer k, does there exist a boolean formula ψ equivalent to φ that has no more than k literals? Stockmeyer [Sto77] considered MEE restricted to formulas in disjunctive normal form, which is denoted by MEE-DNF. Both MEE and MEE-DNF are members of Σ_2^p. Hemaspaandra and Wechsung [HW02] proved that MEE and MEE-DNF are Θ_2^p-hard and that MINIMAL is coNP-hard. Umans [Uma01] showed that MEE-DNF is even Σ_2^p-complete. It is still an open problem to precisely pinpoint the complexity of MEE and of MINIMAL.

- Eiter and Gottlob [EG00] showed that the problem BOUNDED EIGENVECTOR is Σ_2^p-complete, which is defined as follows: Given an $n\times n$ matrix M, an integer eigenvalue λ of M, a subset I of the components, and integers b and z, does there exist a b-bounded $\preceq$-minimal nonzero eigenvector $x = (x_1,\ldots,x_n)$ for λ such that $x_1 = z$? Here, $x \preceq y$ if and only if x and y coincide on the components in I and $\|x\| \leq \|y\|$, where $\|x\| = \left(\sum_{i=1}^n x_i^2\right)^{1/2}$ denotes the L_2 norm of an integer vector x.

  Eiter and Gottlob [EG00] also considered variants of BOUNDED EIGENVECTOR, e.g., by restricting $I = \emptyset$ (i.e., in this restriction one is looking for any shortest eigenvector x among the b-bounded eigenvectors for λ such that $x_1 = z$) and showing that this problem is Δ_2^p-complete in general and Θ_2^p-complete if $b \geq 1$ is a fixed constant.

- Meyer and Stockmeyer [MS72] proved that MINMAX-SAT is Π_2^p-complete, which is defined as follows: Given a boolean formula $\varphi(x,y)$ in conjunctive normal form, with no more than three literals per clause, and a nonnegative integer k, does it hold that for each truth assignment to x there is a truth assignment to y satisfying at least k clauses in $\varphi(x,y)$?

Schaefer and Umans [SU02a, SU02b] provide a comprehensive survey of completeness in the levels of the polynomial hierarchy.

4.2 The Boolean Hierachy over NP

As mentioned in Section 3.2, coNP is the class of problems whose complements are in NP. NP is not known to be closed under complementation;[9] indeed, NP and coNP are widely believed to be distinct classes. Loosely speaking, the levels of the boolean hierarchy over NP consist of those classes whose members result from applying set operations such as complementation, union, and intersection to a finite number of NP sets. In particular, P is the zeroth level of this hierarchy, and NP and coNP constitute its first level. To define the second level of this hierarchy, let

$$\mathrm{DP} = \{A - B \mid A, B \in \mathrm{NP}\}$$

be the class of differences of any two NP sets (equivalently, $\mathrm{DP} = \{A \cap \overline{B} \mid A, B \in \mathrm{NP}\}$), and let

$$\mathrm{coDP} = \{L \mid \overline{L} \in \mathrm{DP}\} = \{\overline{A} \cup B \mid A, B \in \mathrm{NP}\}$$

be the class of problems whose complements are in DP.

The complexity class DP was introduced by Papadimitriou and Yannakakis [PY84] to capture the complexity of problems that are NP-hard or coNP-hard but seemingly not contained in NP or coNP. An example of such a problem—indeed a DP-complete problem (as can be shown essentially analogously to the proof of Cook's Theorem)—is SAT-UNSAT: Given two boolean formulas φ and ψ in conjunctive normal form, is it true that φ is satisfiable but ψ is not satisfiable?

Many more natural problems are known to be DP-complete, for example, exact optimization problems and so-called critical problems. An example of an exact optimization problem is EXACT 4-COLORABILITY: Is the chromatic number[10] of a given graph equal to four? This problem is DP-complete [Rot03] (see [RR06b, RR06a, RR10] for more DP-completeness results on exact optimization problems).

Critical problems are characterized by the feature of either losing or gaining some property due to a smallest possible change in the problem instance. For example, consider the graph minimal uncolorability problem: Given a graph G and a positive integer k, is it true that G is not k-colorable, but removing any one vertex and its incident edges from G makes the resulting graph k-colorable? Cai and Meyer [CM87] proved that this problem is DP-complete.

We have defined the first three levels of the boolean hierarchy over NP to comprise the classes $\mathrm{BH}_0(\mathrm{NP}) = \mathrm{P}$, $\mathrm{BH}_1(\mathrm{NP}) = \mathrm{NP}$, $\mathrm{coBH}_1(\mathrm{NP}) = \mathrm{coNP}$, $\mathrm{BH}_2(\mathrm{NP}) = \mathrm{DP}$, and $\mathrm{coBH}_2(\mathrm{NP}) = \mathrm{coDP}$. Continuing by induction, define the i-th level, $i \geq 3$, of this hierarchy by

$$\begin{aligned} \mathrm{BH}_i(\mathrm{NP}) &= \{A \cup B \mid A \in \mathrm{BH}_{i-2}(\mathrm{NP}), B \in \mathrm{BH}_2(\mathrm{NP})\} \\ \mathrm{coBH}_i(\mathrm{NP}) &= \{\overline{A} \cap \overline{B} \mid A \in \mathrm{BH}_{i-2}(\mathrm{NP}), B \in \mathrm{BH}_2(\mathrm{NP})\}, \end{aligned}$$

and let $\mathrm{BH}(\mathrm{NP}) = \bigcup_{i \geq 1} \mathrm{BH}_i(\mathrm{NP})$.

[9] A class $\mathscr{C}$ of sets is said to be *closed under complementation* if $A \in \mathscr{C}$ implies $\overline{A} \in \mathscr{C}$. Similarly, $\mathscr{C}$ is said to be *closed under union* (respectively, *closed under intersection*) if $A \cup B \in \mathscr{C}$ (respectively, $A \cap B \in \mathscr{C}$) whenever $A \in \mathscr{C}$ and $B \in \mathscr{C}$. It is an easy exercise to show that NP is closed under both union and intersection.

[10] The *chromatic number of graph G* is the smallest number of colors needed to color the vertices of G such that no two adjacent vertices receive the same color. For each $k \geq 3$, the k-COLORABILITY problem, which asks whether the chromatic number of a given graph is at most k (i.e., whether it is k-colorable), is known to be NP-complete [GJ79].

Cai et al. [CGH$^+$88, CGH$^+$89] provided a deep and comprehensive study of the boolean hierarchy over NP. It is known that this hierarchy is contained in the second level of the polynomial hierarchy: $\mathrm{BH}(\mathrm{NP}) \subseteq \Theta_2^p$. Again, it is an open question whether $\mathrm{BH}(\mathrm{NP})$ is a strictly infinite hierarchy or collapses to some finite level. Interestingly, if the boolean hierarchy over NP collapses to some finite level then so does the polynomial hierarchy. The first such result was shown by Kadin [Kad88]: If $\mathrm{BH}_i(\mathrm{NP}) = \mathrm{coBH}_i(\mathrm{NP})$ for some $i \geq 1$,[11] then the polynomial hierarchy collapses to its third level: $\mathrm{PH} = \Sigma_3^p \cap \Pi_3^p$. This result has been improved by showing stronger and stronger collapses of the polynomial hierarchy under the same hypothesis (see the survey [HHH98]). As a final remark, boolean hierarchies over classes other than NP have also been investigated [HR97, BBJ$^+$89].

5 Concluding Remarks

Critics of complexity theory argue that it is not practical to consider only the worst-case complexity; for example, the performance of the simplex method in practice appears to be a fairly low order polynomial complexity, once pathologies like the Klee–Minty polytope are removed. Observations like this motivated Levin [Lev86] to define the notion of average-case complexity (see also Goldreich [Gol97], Impagliazzo [Imp95], and Wang [Wan97a, Wan97b]) and to show that various problems that are hard in the worst case are easy to solve on the average. However, average-case complexity theory has its drawbacks as well; for example, results on the average-case complexity of problems heavily depend on the distribution of inputs used. Up to date, only a handful of problems have been shown to be hard on the average [Lev86, Wan97b]. In this regard, a very interesting result is due to Ajtai and Dwork, who built a public-key cryptosystem whose security rests on the shortest vector problem in lattices, which is as hard in the average case as it is in the worst case [Ajt96, AD97].

Advocates of complexity theory argue that it does provide a practical guide to algorithm design, and this is especially true of *approximation algorithms*. These comprise an approach to "solving" hard problems, where we give up accuracy in order to have tractability. To be an approximation algorithm, it must have polynomial complexity, and it must give a guaranteed bound on the quality of a solution. While many NP-hard problems allow for efficient approximation schemes, other NP-hard problems provably do not (under reasonable complexity-theoretic assumptions), see, e.g., Håstad [Hås99]. The study of *inapproximability* is also a central task in complexity theory.

Other ways of coping with NP-hardness include looking for heuristical algorithms (either algorithms that are efficient but not always correct, or algorithms that are correct but not always efficient) or showing that NP-hard problems are fixed-parameter tractable (see, e.g., Downey and Fellows [DF99], Flum and Grohe [FG06], and Niedermeier [Nie06] and the surveys by Buss and Islam [BI08] and Lindner and Rothe [LR08]).

There are many more important complexity classes and hierarchies between P and PSPACE that could not be discussed in this brief introduction to computational complexity theory, such as probabilistic classes [Gil77, Sim75], counting classes [Val79a, Val79b, For97], unambiguous

[11]Note that $\mathrm{BH}_i(\mathrm{NP}) = \mathrm{coBH}_i(\mathrm{NP})$, $i \geq 1$, is equivalent to the boolean hierarchy over NP collapsing to its i-th level: $\mathrm{BH}_i(\mathrm{NP}) = \mathrm{coBH}_i(\mathrm{NP}) = \mathrm{BH}_{i+1}(\mathrm{NP}) = \mathrm{coBH}_{i+1}(\mathrm{NP}) = \cdots = \mathrm{BH}(\mathrm{NP})$.

classes [Val76, HR97], the Arthur-Merlin hierarchy [BM88], the low hierarchy within NP [Sch83], etc. Readers interested in these complexity classes and hierarchies are referred to the textbooks mentioned in the introduction.

Acknowledgements

We are grateful to Harvey Greenberg for his advice and helpful comments on this text and for his generous permission to use parts of his original supplement on complexity.

References

[AD97] M. Ajtai and C. Dwork. A public-key cryptosystem with worst-case/average-case equivalence. In *Proceedings of the 29th ACM Symposium on Theory of Computing*, pages 284–293. ACM Press, 1997.

[Ajt96] M. Ajtai. Generating hard instances of lattice problems. In *Proceedings of the 28th ACM Symposium on Theory of Computing*, pages 99–108. ACM Press, 1996.

[BBJ$^+$89] A. Bertoni, D. Bruschi, D. Joseph, M. Sitharam, and P. Young. Generalized boolean hierarchies and boolean hierarchies over RP. In *Proceedings of the 7th Conference on Fundamentals of Computation Theory*, pages 35–46. Springer-Verlag *Lecture Notes in Computer Science #380,* August 1989.

[BI08] J. Buss and T. Islam. The complexity of fixed-parameter problems. *SIGACT News*, 39(1):34–46, March 2008.

[BM88] L. Babai and S. Moran. Arthur-Merlin games: A randomized proof system, and a hierarchy of complexity classes. *Journal of Computer and System Sciences*, 36(2):254–276, 1988.

[CGH$^+$88] J. Cai, T. Gundermann, J. Hartmanis, L. Hemachandra, V. Sewelson, K. Wagner, and G. Wechsung. The boolean hierarchy I: Structural properties. *SIAM Journal on Computing*, 17(6):1232–1252, 1988.

[CGH$^+$89] J. Cai, T. Gundermann, J. Hartmanis, L. Hemachandra, V. Sewelson, K. Wagner, and G. Wechsung. The boolean hierarchy II: Applications. *SIAM Journal on Computing*, 18(1):95–111, 1989.

[CM87] J. Cai and G. Meyer. Graph minimal uncolorability is D^{P}-complete. *SIAM Journal on Computing*, 16(2):259–277, 1987.

[Coo71] S. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd ACM Symposium on Theory of Computing*, pages 151–158. ACM Press, 1971.

[DF99] R. Downey and M. Fellows. *Parameterized Complexity*. Springer-Verlag, 1999.

[EG00] T. Eiter and G. Gottlob. Complexity results for some eigenvector problems. *International Journal of Computer Mathematics*, 76(1/2):59–74, 2000.

[FG06] J. Flum and M. Grohe. *Parameterized Complexity Theory*. EATCS Texts in Theoretical Computer Science. Springer-Verlag, 2006.

[For97] L. Fortnow. Counting complexity. In L. Hemaspaandra and A. Selman, editors, *Complexity Theory Retrospective II*, pages 81–107. Springer-Verlag, 1997.

[GB95] F. Glover and D. Babayev. New results for aggregating integer-valued equations. *Annals of Operations Research*, 58(3):227–242, 1995.

[Gil77] J. Gill. Computational complexity of probabilistic Turing machines. *SIAM Journal on Computing*, 6(4):675–695, 1977.

[GJ79] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[Gol97] O. Goldreich. Notes on Levin's theory of average-case complexity. Technical Report TR97-058, Electronic Colloquium on Computational Complexity, November 1997.

[Gre01] H. Greenberg. Computational complexity. In A. Holder, editor, *Mathematical Programming Glossary*. INFORMS Computing Society, May 2001.

[Hås99] J. Håstad. Clique is hard to approximate within $n^{1-\varepsilon}$. *Acta Mathematica*, 182(1):105–142, 1999.

[HHH98] E. Hemaspaandra, L. Hemaspaandra, and H. Hempel. What's up with downward collapse: Using the easy-hard technique to link boolean and polynomial hierarchy collapses. *SIGACT News*, 29(3):10–22, 1998.

[HHR97] E. Hemaspaandra, L. Hemaspaandra, and J. Rothe. Exact analysis of Dodgson elections: Lewis Carroll's 1876 voting system is complete for parallel access to NP. *Journal of the ACM*, 44(6):806–825, 1997.

[HLS65] J. Hartmanis, P. Lewis, and R. Stearns. Classification of computations by time and memory requirements. In *Proceedings of the IFIP Congress 65*, pages 31–35. International Federation for Information Processing, Spartan Books, 1965.

[HMU01] J. Hopcroft, R. Motwani, and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, second edition, 2001.

[HR97] L. Hemaspaandra and J. Rothe. Unambiguous computation: Boolean hierarchies and sparse Turing-complete sets. *SIAM Journal on Computing*, 26(3):634–653, June 1997.

[HS65] J. Hartmanis and R. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.

[HU79] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[HW02] E. Hemaspaandra and G. Wechsung. The minimization problem for boolean formulas. *SIAM Journal on Computing*, 31(6):1948–1958, 2002.

[Imp95] R. Impagliazzo. A personal view of average-case complexity. In *Proceedings of the 10th Structure in Complexity Theory Conference*, pages 134–147. IEEE Computer Society Press, 1995.

[Joh81] D. Johnson. The NP-completeness column: An ongoing guide. *Journal of Algorithms*, 2(4):393–405, December 1981. First column in a series of columns on NP-completeness appearing in the same journal.

[Kad88] J. Kadin. The polynomial time hierarchy collapses if the boolean hierarchy collapses. *SIAM Journal on Computing*, 17(6):1263–1282, 1988. Erratum appears in the same journal, 20(2):404, 1991.

[Kar72] R. Karp. Reducibilities among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103, 1972.

[KPP04] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer-Verlag, Berlin, Heidelberg, New York, 2004.

[Lev73] L. Levin. Universal sorting problems. *Problemy Peredaci Informacii*, 9:115–116, 1973. In Russian. English translation in *Problems of Information Transmission*, 9:265–266, 1973.

[Lev86] L. Levin. Average case complete problems. *SIAM Journal on Computing*, 15(1):285–286, 1986.

[LR08] C. Lindner and J. Rothe. Fixed-parameter tractability and parameterized complexity, applied to problems from computational social choice. In A. Holder, editor, *Mathematical Programming Glossary*. INFORMS Computing Society, October 2008.

[LSH65] P. Lewis, R. Stearns, and J. Hartmanis. Memory bounds for recognition of context-free and context-sensitive languages. In *Proceedings of the 6th IEEE Symposium on Switching Circuit Theory and Logical Design*, pages 191–202, 1965.

[MS72] A. Meyer and L. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *Proceedings of the 13th IEEE Symposium on Switching and Automata Theory*, pages 125–129, 1972.

[Nie06] R. Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford University Press, 2006.

[Odi89] P. Odifreddi. *Classical Recursion Theory*. North-Holland, 1989.

[Pap94] C. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[PY84] C. Papadimitriou and M. Yannakakis. The complexity of facets (and some facets of complexity). *Journal of Computer and System Sciences*, 28(2):244–259, 1984.

[Rog67] H. Rogers, Jr. *The Theory of Recursive Functions and Effective Computability*. McGraw-Hill, 1967.

[Rot03] J. Rothe. Exact complexity of Exact-Four-Colorability. *Information Processing Letters*, 87(1):7–12, 2003.

[Rot05] J. Rothe. *Complexity Theory and Cryptology. An Introduction to Cryptocomplexity*. EATCS Texts in Theoretical Computer Science. Springer-Verlag, 2005.

[RR06a] T. Riege and J. Rothe. Completeness in the boolean hierarchy: Exact-Four-Colorability, minimal graph uncolorability, and exact domatic number problems – a survey. *Journal of Universal Computer Science*, 12(5):551–578, 2006.

[RR06b] T. Riege and J. Rothe. Complexity of the exact domatic number problem and of the exact conveyor flow shop problem. *Theory of Computing Systems*, 39(5):635–668, September 2006.

[RR10] M. Roos and J. Rothe. Complexity of social welfare optimization in multiagent resource allocation. In *Proceedings of the 9th International Joint Conference on Autonomous Agents and Multiagent Systems*. IFAAMAS, May 2010. To appear.

[Sch83] U. Schöning. A low and a high hierarchy within NP. *Journal of Computer and System Sciences*, 27:14–28, 1983.

[Sel94] A. Selman. A taxonomy of complexity classes of functions. *Journal of Computer and System Sciences*, 48(2):357–381, 1994.

[SHL65] R. Stearns, J. Hartmanis, and P. Lewis. Hierarchies of memory limited computations. In *Proceedings of the 6th IEEE Symposium on Switching Circuit Theory and Logical Design*, pages 179–190, 1965.

[Sim75] J. Simon. *On Some Central Problems in Computational Complexity*. PhD thesis, Cornell University, Ithaca, NY, January 1975. Available as Cornell Department of Computer Science Technical Report TR75-224.

[Sto77] L. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3(1):1–22, 1977.

[SU02a] M. Schaefer and C. Umans. Completeness in the polynomial-time hierarchy: Part I: A compendium. *SIGACT News*, 33(3):32–49, September 2002.

[SU02b] M. Schaefer and C. Umans. Completeness in the polynomial-time hierarchy: Part II. *SIGACT News*, 33(4):22–36, December 2002.

[Tov02] C. Tovey. Tutorial on computational complexity. *Interfaces*, 32(3):30–61, 2002.

[Tur36] A. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, ser. 2*, 42:230–265, 1936. Correction, *ibid*, vol. 43, pp. 544–546, 1937.

[Uma01] C. Umans. The minimum equivalent DNF problem and shortest implicants. *Journal of Computer and System Sciences*, 63(4):597–611, 2001.

[Val76] L. Valiant. The relative complexity of checking and evaluating. *Information Processing Letters*, 5(1):20–23, 1976.

[Val79a] L. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2):189–201, 1979.

[Val79b] L. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.

[Wag87] K. Wagner. More complicated questions about maxima and minima, and some closures of NP. *Theoretical Computer Science*, 51:53–80, 1987.

[Wan97a] J. Wang. Average-case computational complexity theory. In L. Hemaspaandra and A. Selman, editors, *Complexity Theory Retrospective II*, pages 295–328. Springer-Verlag, 1997.

[Wan97b] J. Wang. Average-case intractable NP problems. In D. Du and K. Ko, editors, *Advances in Languages, Algorithms, and Complexity*, pages 313–378. Kluwer Academic Publishers, 1997.

[Wil94] H. Wilf. *Algorithms and Complexity*. A. K. Peters, 2nd edition, 1994.

[Wra77] C. Wrathall. Complete sets and the polynomial-time hierarchy. *Theoretical Computer Science*, 3:23–33, 1977.