Chapter 1

The Gödel Phenomena in Mathematics: A Modern View

Avi Wigderson

Herbert Maass Professor
School of Mathematics
Institute for Advanced Study
Princeton, New Jersey, USA

1.1 Introduction

What are the limits of mathematical knowledge? The purpose of this chapter is to introduce the main concepts from *computational complexity theory* that are relevant to algorithmic accessibility of mathematical understanding. In particular, I'll discuss the $\mathcal{P}$ versus $\mathcal{NP}$ problem, its possible impact on research in mathematics, and how interested Gödel himself was in this computational viewpoint.

Much of the technical material will be necessarily sketchy. The interested reader is referred to the standard texts on computational complexity theory, primarily [5, 25, 43, 61].

1.1.1 Overview

Hilbert believed that all mathematical truths are *knowable* and set the threshold for mathematical knowledge at the ability to devise a "mechanical procedure." This dream was shattered by Gödel and Turing. Gödel's incompleteness theorem exhibited true statements that can never be proved. Turing formalized Hilbert's notion of computation and of finite algorithms (thereby initiating the computer revolution) and proved that some problems are *undecidable*—they have no such algorithms.

While the first examples of such "unknowables" seemed somewhat unnatural, more and more natural examples of unprovable or undecidable problems were found in different areas of mathematics. The independence of the continuum hypothesis and the undecidability of Diophantine equations are famous

early examples. This became known as the *Gödel phenomena*, and its effect on the practice of mathematics has been debated since. Many argued that while some of the inaccessible truths above are natural, they are far from what is really of interest to most working mathematicians. Indeed, it would seem that in the seventy-five years since the incompleteness theorem, mathematics has continued thriving, with remarkable achievements such as the recent settlement of Fermat's last "theorem" by Wiles and the Poincaré conjecture by Perelman. So, are there *interesting* mathematical truths that are *unknowable*?

The main point of this chapter is that when *knowability* is interpreted by modern standards, namely via *computational complexity*, the Gödel phenomena are very much with us. We argue that to understand a mathematical structure, having a *decision procedure* is but a first approximation, and that a real understanding requires an *efficient* algorithm. Remarkably, Gödel was the first to propose this modern view, in a letter to von Neumann in 1956, which was discovered only in the 1990s.

Meanwhile, from the mid-1960s on, the field of theoretical computer science has made formal Gödel's challenge and has created a theory that enables quantification of the difficulty of *computational* problems. In particular, a reasonable way to capture *knowable* problems (which we can efficiently solve) is the class $\mathcal{P}$, and a reasonable way to capture *interesting* problems (which we would like to solve) is the class $\mathcal{NP}$. Moreover, assuming the widely believed $\mathcal{P} \neq \mathcal{NP}$ conjecture, the class $\mathcal{NP}$-complete captures interesting *unknowable* problems.

In this chapter, I define these complexity classes, explain their intuitive meaning, and show how Gödel foresaw some of these definitions and connections. I relate computational difficulty to mathematical difficulty and argue how such notions, developed in computational complexity, may explain the difficulty mathematicians have in accessing structural, non-computational information. I also survey proof complexity—the study of lengths of proofs in different proof systems.

Finally, I point out that this modern view of the limits of mathematical knowledge is adopted by a growing number of mathematicians, working on a diverse set of interesting structures in different areas of mathematics. This activity increases interaction with computer scientists and benefits both fields. Results are of two types, as in standard computer science problems. On the one hand, there is a growing effort to go beyond characterizations of mathematical structures, and attempts are made to provide efficient recognition algorithms. On the other, there is a growing number of $\mathcal{NP}$-completeness results, providing the *stamp of difficulty* for achieving useful characterizations and algorithms. This second phenomenon, now ubiquitous in science and mathematics, may perhaps better capture Gödel's legacy on what is unknowable in mathematics.

1.1.2 Decision Problems and Finite Algorithms

Which mathematical structures can we hope to understand? Let us focus on the most basic mathematical task of *classification*. We are interested in a particular class of objects and a particular property. We seek to *understand* which of the objects have the property and which do not. Let us consider the following

examples:

(1) Which valid sentences in first-order predicate logic are provable?

(2) Which Diophantine equations have solutions?

(3) Which knots are unknotted?

(4) Which elementary statements about the reals are true?

It is clear that each object from the families above has a finite representation. Hilbert's challenge was about the possibility to find, for each such family, a finite procedure that would solve the decision problem *in finite time* for every object in the family. In his seminal paper [63, 64], Turing formulated the *Turing machine*, a mathematical definition of an *algorithm*, capturing such finite mechanical procedures. This allowed a mathematical study of Hilbert's challenge.

Hilbert's Entscheidungsproblem—problem (1) above—was the first to be resolved, in the same paper by Turing. Turing showed that problem (1) is undecidable, namely there is no algorithm that distinguishes provable from unprovable statements of first-order predicate logic. Hilbert's 10th problem—problem (2) above—was shown to be undecidable as well, in a series of works by Davis, Putnam, Robinson, and Matiasevich [41]. Again, no algorithm can distinguish solvable from unsolvable Diophantine equations.

The crucial ingredient in those (and all other) undecidability results is showing that each of these mathematical structures can *encode computation*. This is known today to hold for many different structures in algebra, topology, geometry, analysis, logic, and more, even though a priori the structures studied seem to be completely unrelated to computation. It is as though much of contemporary work in mathematics is pursuing, unwittingly and subconsciously, an agenda with essential computational components. I shall return to refined versions of this idea later.

The notion of a decision procedure as a minimal requirement for understanding of a mathematical problem has also led to direct positive results. It suggests that we look for a decision procedure as *a means*, or as *a first step* for understanding a problem. For problems (3) and (4) above this was successful. Haken [30] showed how knots can be so understood, with his decision procedure for problem (3), and Tarski [62] showed that real-closed fields can be understood with a decision procedure for problem (4). Naturally, significant *mathematical, structural* understanding was needed to develop these algorithms. Haken developed the theory of *normal surfaces* and Tarski invented *quantifier elimination* for their algorithms, both cornerstones of the respective fields.

This reveals only the obvious: mathematical and algorithmic understanding are related and often go hand in hand. There are many ancient examples. The earliest is probably Euclid's greatest common divisor (GCD) algorithm. Abel's proof that roots of real polynomials of degree at least 5 have no *formula* with radicals is the first impossibility result (of certain classes of algorithms). Of course, Newton's *algorithm* for approximating such roots is a satisfactory

practical alternative. What was true in previous centuries is truer in this one: the language of algorithms is slowly becoming competitive with the language of equations and formulas (which are special cases of algorithms) for explaining complex mathematical structures.

Back to our four problems. The undecidability of problems (1) and (2) certainly suggests that these structures cannot be mathematically understood in general. This led researchers to consider special cases of them that are decidable. But does decidability—for example, that of problems (3) and (4)—mean that we completely understand these structures? Far from it. Both algorithms are extremely complex and time consuming. Even structurally speaking, there are no known complete knot invariants or characterizations of real varieties. How can one explain such mathematical difficulties for decidable problems?

It is natural to go beyond decidability and try to quantify the level of understanding. We will use a computational yardstick for it. We argue that better mathematical understanding goes hand in hand with better algorithms for "obtaining" that understanding from the given structures. To formalize this, we will introduce the computational terms that are central to the theory of computational complexity. There is no better way to start this than with Gödel's letter to von Neumann.

1.2 Gödel's Letter to von Neumann

In Gödel's letter, which I include below in its entirety for completeness, complexity is discussed only in the second paragraph—it is remarkable how much this paragraph contains! It defines *asymptotic* and *worst-case* time complexity, suggests the study of the *satisfiability* problem, discusses the mathematical significance of having a fast algorithm for it, and even speculates on the possibility of the existence of such an algorithm. In the section that follows the letter, I identify the letter's contributions and insights in modern language.

1.2.1 The Letter

First, a bit on the context. The letter was written when von Neumann was in the hospital, already terminally ill with cancer (he died a year later). It was written in German, and here we reproduce a translation. The surveys by Sipser [60] and Hartmanis [31] provide the original letter, as well as more details on the translation.

As far as we know, the discussion Gödel was trying to initiate never continued. Moreover, we have no evidence of Gödel's thinking more about the subject. Again, we will refer later only to the second paragraph, which addresses the computational complexity issue.

```
Princeton, 20 March 1956

Dear Mr.  von Neumann:
```

With the greatest sorrow I have learned of your illness. The news came to me as quite unexpected. Morgenstern already last summer told me of a bout of weakness you once had, but at that time he thought that this was not of any greater significance. As I hear, in the last months you have undergone a radical treatment and I am happy that this treatment was successful as desired, and that you are now doing better. I hope and wish for you that your condition will soon improve even more and that the newest medical discoveries, if possible, will lead to a complete recovery.

Since you now, as I hear, are feeling stronger, I would like to allow myself to write you about a mathematical problem, of which your opinion would very much interest me: One can obviously easily construct a Turing machine, which for every formula F in first order predicate logic and every natural number n, allows one to decide if there is a proof of F of length n (length = number of symbols). Let $\psi(F,n)$ be the number of steps the machine requires for this and let $\phi(n) = \max_F \psi(F,n)$. The question is how fast $\phi(n)$ grows for an optimal machine. One can show that $\phi(n) \geq k \cdot n$. If there really were a machine with $\phi(n) \approx k \cdot n$ (or even $\phi(n) \approx k \cdot n^2$), this would have consequences of the greatest importance. Namely, it would obviously mean that in spite of the undecidability of the Entscheidungsproblem, the mental work of a mathematician concerning Yes-or-No questions (apart from the postulation of axioms) could be completely replaced by a machine. After all, one would simply have to choose the natural number n so large that when the machine does not deliver a result, it makes no sense to think more about the problem. Now it seems to me, however, to be completely within the realm of possibility that $\phi(n)$ grows that slowly. Since 1.) it seems that $\phi(n) \geq k \cdot n$ is the only estimation which one can obtain by a generalization of the proof of the undecidability of the Entscheidungsproblem and 2.) after all $\phi(n) \approx k \cdot n$ (or $\phi(n) \approx k \cdot n^2$) only means that the number of steps as opposed to trial and error can be reduced from N to $\log N$ (or $(\log N)^2$). However, such strong reductions appear in other finite problems, for example in the computation of the quadratic residue symbol using repeated application of the law of reciprocity. It would be interesting to know, for instance, the situation concerning the determination of primality of a number and how strongly in general the number of steps in finite combinatorial problems can be reduced with respect to simple exhaustive search.

> I do not know if you have heard that "Post's problem," whether there are degrees of unsolvability among problems of the form $(\exists y)\phi(y, x)$, where ϕ is recursive, has been solved in the positive sense by a very young man by the name of Richard Friedberg. The solution is very elegant. Unfortunately, Friedberg does not intend to study mathematics, but rather medicine (apparently under the influence of his father). By the way, what do you think of the attempts to build the foundations of analysis on ramified type theory, which have recently gained momentum? You are probably aware that Paul Lorenzen has pushed ahead with this approach to the theory of Lebesgue measure. However, I believe that in important parts of analysis non-eliminable impredicative proof methods do appear.
>
> I would be very happy to hear something from you personally. Please let me know if there is something that I can do for you. With my best greetings and wishes, as well to your wife,
>
> Sincerely yours,
>
> Kurt Gödel
>
> P.S. I heartily congratulate you on the award that the American government has given to you.

1.2.2 Time Complexity and Gödel's Foresight

In this section, I go through the main ingredients of Gödel's letter, most of which were independently[1] identified and developed in the evolution of computational complexity in the 1960s and 1970s. These include the basic model, input representation, (asymptotic, worst-case) time complexity,[2] brute-force algorithms and the possibility of beating them, proving lower bounds, and last but not least, Gödel's choice of a focus problem and its significance. I comment on the remarkable foresight of some of Gödel's choices. When introducing some of these notions, I use common, modern notation.

- **Computability vs. complexity:** All problems we will discuss here are computable—that is, they have a finite algorithm. Gödel states that for his problem, "one can...easily construct a Turing machine." We take for granted that the given problem is decidable and ask for the complexity of the *best* or *optimal* algorithm. Gödel is the first on record to suggest shifting focus from computability to complexity.

[1] Recall that the letter was discovered only in the 1990s.

[2] I shall focus on time as the primary resource of algorithms when studying their efficiency. Other resources, such as memory, parallelism, and more, are studied in computational complexity, but I will not treat them here.

- **The model:** Gödel's computational model of choice is the Turing machine. Time will be measured as the number of elementary steps on a Turing machine. We note that this is a nontrivial choice for a discussion of time complexity back in 1956. The Church-Turing thesis of the equivalence of all feasible computational models was around, and all known algorithmic models were known to simulate each other. However, that thesis did not include time bounds, and it seems that Gödel takes for granted (or knows) that all known models can simulate each other *efficiently* (something that is now well established).

- **The problem:** Gödel focuses on one problem to define complexity, a finite version of the Entscheidungsproblem, in which the task is to determine if a formula F has a proof of length n. This is by no means an arbitrary choice, and Gödel is well aware of it, as will be discussed below. Put differently, Gödel's problem asks if we can *satisfy* a first-order logic verifier that F is provable using only n symbols. We shall later meet its cousin, the problem SAT (abbreviating "SATisfyability"), in which the task is to determine if a *propositional* formula has a satisfying assignment. It is not hard to see that SAT captures Gödel's problem,[3] and so we will call it SAT.

- **Input representation:** Every finite object (formulas, integers, graphs, etc.) can be represented as a binary string. We let I stand for the set of all finite binary strings. Let f be a decision problem ("`Yes-or-No question`" in Gödel's letter), like those in Section 1.1.2. Then $f : \mathrm{I} \to \{0, 1\}$ is what we are trying to compute. Thus, we consider Turing machines that for every input x halt with the answer $f(x)$. In Gödel's problem the input is the pair (F, n), and the task is computing whether the first-order formula F has a proof of length at most n.

- **Input length:** This is always taken in computational complexity to be the binary length of the input. Clearly Gödel makes the same choice: when he talks about the complexity of testing primality of an integer N, he takes the input length to be $\log N$, the number of bits needed to describe N. Of course, finite objects may have many representations, which differ in length, and one must pick "reasonable" encodings [4].

- **Time complexity:** Complexity is measured as a function of *input length*. Adapting Gödel's notation, we fix any Turing machine M computing f. We let $\phi(n)$ denote the maximum,[5] over all inputs x of length n, of the number of steps M takes when computing $f(x)$. We then consider the

[3] Simply assign propositional variables to the n possible proof symbols, and encode the verification process so that indeed they form a proof of F as a Boolean formula.

[4] For example, in Gödel's problem it is possible to encode the input using $|F| + \log n$ bits, where $|F|$ is the encoding length (in bits) of the given first-order formula. However, as we will later see, it is more natural to take it to be $|F| + n$, allowing for n variables to encode the possible values of the potential proof symbols. For this problem Gödel indeed measures complexity as a function of n.

[5] This worst-case analysis is the most common in complexity theory, though of course other measures are important and well studied too. In particular, average-case analysis (typical

growth of $\phi(n)$ for the "`optimal machine`"[6] M. It is quite remarkable that Gödel selects both the asymptotic viewpoint and worst-case complexity—neither is an obvious choice but both proved extremely fruitful in the development of computational complexity.

- **Brute-force algorithms:** All problems mentioned by Gödel—SAT, primality testing, and computing quadratic residue—have what Gödel calls "`trial and error`" and "`simple exhaustive search`" algorithms. All these problems happen to be in the class $\mathcal{NP}$, and these obvious trivial algorithms require exponential time (in the input length).

- **Efficient algorithms:** Gödel specifically suggests that time complexity[7] $O(n)$ or $O(n^2)$ is efficient (or tractable) enough for practical purposes. This did not change for fifty years, despite enormous changes in computer speed, and is a gold standard of efficiency. I'll try to explain why computational complexity theory chose to call (any) polynomial-time algorithms efficient and the class $\mathcal{P}$ to include all problems with such algorithms.

- **The complexity of SAT:** Gödel is completely aware of what it would mean if the problem SAT has an efficient algorithm: "`the mental work of a mathematician concerning Yes-or-No questions (apart from the postulation of axioms) could be completely replaced by a machine`." So he realizes that solving this problem will solve numerous others in mathematics. While not stating precisely that it is $\mathcal{NP}$-complete ("captures" all problems in $\mathcal{NP}$), he does ask "`how strongly in general the number of steps in finite combinatorial problems can be reduced with respect to simple exhaustive search`," a set of problems we naturally identify now with $\mathcal{NP}$.

- **Gödel's speculation:** The letter states "`Now it seems to me...to be completely within the realm of possibility that` $\phi(n)$ `grows that slowly`," namely that $SAT \in \mathcal{P}$, which as we shall see implies $\mathcal{P} = \mathcal{NP}$. Gödel suggests this possibility based on the existence of highly nontrivial efficient algorithms that improve exponentially over brute-force search: "`such strong reductions appear in other finite problems, for example in the computation of the quadratic residue symbol using repeated application of the law of reciprocity`" (the trivial exponential time algorithm would rely on factoring, while Gauss's reciprocity allows a GCD-like polynomial-time algorithm.[8]) This lesson, that there are sometimes ingenious ways of cutting the exponential search space for many important problems, was taught again and again in the last half-century.

behavior of algorithms under natural input distributions) is central both for algorithm design and for cryptography.

[6] Strictly speaking, this is not well defined.

[7] We use standard asymptotic notation: $g(n) = O(h(n))$ if for some constant k and for all n we have $g(n) \leq kh(n)$. In this case we also say that $h(n) = \Omega(g(n))$.

[8] In any reasonable implementation Gödel had in mind, this would take time n^3, which shows that Gödel may have indeed considered the class $\mathcal{P}$ for efficiently solved problems.

- **Lower bounds:** Gödel tried proving that the complexity of SAT, $\phi(n)$, cannot grow too slowly. But he could only prove a linear lower bound[9] $\phi(n) = \Omega(n)$. And, for Turing machines, not much has changed in this half century—we still have no superlinear lower bounds. Proving such lower bounds, namely proving that computational problems are intrinsically difficult, is a central task of complexity. I shall discuss partial success as well as the difficulty of this task later.

The next sections elaborate on these and other notions and issues and provide some historical background on their development in computational complexity theory. I first deal with the complexity of computation, and then move on to discuss the complexity of proofs.

1.3 Complexity Classes, Reductions and Completeness

In this section, I define efficient computations, efficient reductions between problems, efficient verification, and the classes $\mathcal{P}$, $\mathcal{NP}$, co$\mathcal{NP}$, and $\mathcal{NP}$-complete. I will keep referring back to Gödel's letter.

1.3.1 Efficient Computation and the Class $\mathcal{P}$

In all that follows, I focus on asymptotic complexity and analyze time spent on a problem as a function of input length. The asymptotic viewpoint is inherent to computational complexity theory. We shall see in this chapter that it reveals structure that would probably be obscured by finite, precise analysis.

Efficient computation (for a given problem) will be taken to be one whose runtime on any input of length n is bounded by a *polynomial* function in n. Let I_n denote all binary sequences in I of length n.

Definition 1.3.1 (**The class $\mathcal{P}$**)**.** A function $f\colon \mathrm{I} \to \mathrm{I}$ is in the class $\mathcal{P}$ if there is an algorithm computing f and positive constants A, c such that for every n and every $x \in \mathrm{I}_n$ the algorithm computes $f(x)$ in at most An^c steps.

Note that the definition applies in particular to Boolean functions (whose output is $\{0, 1\}$), which capture classification problems. For convenience, we will sometimes think of $\mathcal{P}$ as the class containing *only* these classification problems. Observe that a function with a long output can be viewed as a sequence of Boolean functions, one for each output bit.

This definition was suggested by Cobham [15], Edmonds [21], and Rabin [51], all attempting to formally delineate *efficient* from just finite (in their cases, exponential time) algorithms. Of course, nontrivial polynomial-time algorithms were discovered earlier, long before the computer age. Many were discovered by mathematicians, who needed efficient methods to calculate (by

[9] Note that while this lower bound is trivial in the modern formulation of SAT, in which the propositional formula has length at least n, it is far from trivial in Gödel's formulation. Gödel does not supply a proof, and Buss provides one in [11].

hand). The most ancient and famous example is Euclid's GCD algorithm, which bypasses the need to factor the inputs when computing their greatest common factor.

Why polynomial? The choice of polynomial time to represent efficient computation seems arbitrary, and different possible choices can be made.[10] However, this particular choice was extremely useful and has justified itself over time from many points of view. We list some important ones.

Polynomials typify "slowly growing" functions. The closure of polynomials under addition, multiplication, and composition preserves the notion of efficiency under natural programming practices, such as using two programs in sequence, or using one as a subroutine of another. This choice removes the necessity to describe the computational model precisely (e.g., it does not matter if we allow arithmetic operations only on single digits or on arbitrary integers, since long addition, subtraction, multiplication, and division have simple polynomial-time algorithms taught in grade school). Similarly, we need not worry about data representation: one can efficiently translate between essentially any two natural representations of a set of finite objects.

¿From a practical viewpoint, while a running time of, say, n^2 is far more desirable than n^{100}, very few known efficient algorithms for natural problems have exponents above 3 or 4. On the other hand, many important natural problems that so far resist efficient algorithms cannot at present be solved faster than in *exponential* time. Thus reducing their complexity to (any) polynomial is a huge conceptual improvement (and when this is achieved, often further reductions of the exponent are found).

The importance of understanding the class $\mathcal{P}$ is obvious. There are numerous computational problems that arise (in theory and practice) that demand efficient solutions. Many algorithmic techniques were developed in the past four decades and enable solving many of these problems (see, for example, the textbook [18]). These drive the ultra-fast home computer applications we now take for granted like web searching, spell checking, data processing, computer game graphics, and fast arithmetic, as well as heavier-duty programs used across industry, business, math, and science. But many more problems yet (some of which we shall meet soon), perhaps of higher practical and theoretical value, remain elusive. The challenge of *characterizing* this fundamental mathematical object—the class $\mathcal{P}$ of efficiently solvable problems—is far beyond us at this point.

I end this section with a few examples of nontrivial problems in $\mathcal{P}$ of mathematical significance. In each the interplay of mathematical and computational understanding needed for the development of these algorithms is evident.

- **Primality testing.** Given an integer, determine if it is prime. Gauss challenged the mathematical community to find an efficient algorithm, but it took two centuries to solve. The story of this recent achievement of [3] and its history are beautifully recounted in [26].

[10] And indeed such choices are studied in computational complexity.

- **Linear programming.** Given a set of linear inequalities in many variables, determine if they are mutually consistent. This problem and its optimization version capture numerous others (finding optimal strategies of a zero-sum game is one), and the convex optimization techniques used to give the efficient algorithms [38, 35] for it do much more (see, for example, [58].)

- **Factoring polynomials.** Given a multivariate polynomial with *rational* coefficients, find its irreducible factors over $\mathbb{Q}$. Again, the tools developed in [39] (mainly regarding "short" bases in lattices in $\mathbb{R}^n$) have numerous other applications.

- **Hereditary graph properties.** Given a finite graph, test if it can be embedded on a fixed surface (like the plane or the torus). A vastly more general result is known, namely testing *any* hereditary property (one that is closed under vertex removal and edge contraction). It follows the monumental structure theory [56] of such properties, including a *finite basis theorem* and its algorithmic versions.[11]

- **Hyperbolic word problem.** Given any presentation of a hyperbolic group by generators and relations, and a word w in the generators, determine if w represents the identity element. The techniques give isoperimetric bounds on the Cayley graphs of such groups and more [27].

1.3.2 Efficient Verification and the Class $\mathcal{NP}$

Now we change our view of classification problems from classifying functions to classifying sets. We shall see that some subtle (and highly relevant mathematical) aspects of complexity come to life when we focus on classifying sets. Thus, a classification problem will be any subset $C \subset \mathrm{I}$. It is convenient for this section to view C as defining a property; $x \in C$ are objects having the property, and $x \notin C$ are objects that do not. While this is formally equivalent to having a Boolean function $f : \mathrm{I} \to \{0, 1\}$ that gives opposite values on C and $\mathrm{I} \setminus C$, this view allows us to distinguish between the complexity of these two sets.

We are given an input $x \in \mathrm{I}$ (describing a mathematical object) and are supposed to determine if $x \in C$ or not. If we had an efficient algorithm for C, we could simply apply it to x. But if we don't, what is the next best thing? One answer is: a *convincing proof* that $x \in C$. Before defining it formally, let us see a motivating example.

As Gödel points out, the working mathematician meets such examples daily, reading a typical math journal paper. In it, we typically find a (claimed) theorem, followed by an (alleged) proof. Thus, we are verifying claims of the type $x \in$ *THEOREMS*, where *THEOREMS* is the set of all provable statements in, say, set theory. It is taken for granted that the written proof is *short* (page limit)

[11] To be fair, while running in polynomial time, the algorithms here have huge exponents. We expect that further, deeper structural understanding will be needed for more efficient algorithms.

and *easily verifiable* (otherwise the referee/editor would demand clarifications), regardless of how long it took to discover.

The class $\mathcal{NP}$ contains all properties C for which membership (namely statements of the form $x \in C$) have *short, efficiently verifiable* proofs. As before, we use polynomials to define both terms. A candidate proof y for the claim $x \in C$ must have length at most polynomial in the length of x. And the verification that y indeed proves this claim must be checkable in polynomial time. Finally, if $x \notin C$, no such y should exist.

Definition 1.3.2 (**The class $\mathcal{NP}$**)**.** The set C is in the class $\mathcal{NP}$ if there is a function $V_C \in \mathcal{P}$ and a constant k such that

- If $x \in C$ then $\exists y$ with $|y| \leq |x|^k$ and $V_C(x, y) = 1$
- If $x \notin C$ then $\forall y$ we have $V_C(x, y) = 0$

Thus each set C in $\mathcal{NP}$ may be viewed as a set of theorems in the complete and sound proof system defined by the verification process V_C.

A sequence y that "convinces" V_C that $x \in C$ is often called a *witness* or *certificate* for the membership of x in C. Again, we stress that the definition of $\mathcal{NP}$ is not concerned with how difficult it is to come up with a witness y. Indeed, the acronym $\mathcal{NP}$ stands for "nondeterministic polynomial time," where the nondeterminism captures the ability of a *hypothetical* "nondeterministic" machine to "guess" a witness y (if one exists) and then verify it deterministically.

Nonetheless, the complexity of finding a witness is of course important, as it captures the *search problem* associated with $\mathcal{NP}$ sets. Every decision problem C (indeed every verifier V_C for C) in $\mathcal{NP}$ comes with a natural search problem associated with it: Given $x \in C$, *find* a short witness y that "convinces" V_C. A correct solution to this search problem can be easily verified by V_C.

While it is usually the search problems that occupy us, from a computational standpoint it is often more convenient to study the decision versions. Almost always both versions are equivalent.[12]

These definitions of $\mathcal{NP}$ were first given (independently and in slightly different forms) by Cook [16] and Levin [40], although Edmonds discusses "good characterization" that captures the essence of efficient verification already in [20]. There is much more to these seminal papers than this definition, and we shall discuss it later at length.

It is evident that decision problems in $\mathcal{P}$ are also in $\mathcal{NP}$. The verifier V_C is simply taken to be the efficient algorithm for C, and the witness y can be the empty sequence.

Corollary 1.3.3. *$\mathcal{P} \subseteq \mathcal{NP}$.*

A final comment is that problems in $\mathcal{NP}$ have trivial *exponential time* algorithms. Such algorithms search through all possible short witnesses and try to

[12] A notable possible exception is the set *COMPOSITES* (with a verification procedure that accepts as witness a nontrivial factor). Note that while *COMPOSITES* $\in \mathcal{P}$ as a decision problem, the related search problem is equivalent to Integer Factorization, which is not known to have an efficient algorithm.

verify each. Now we can make Gödel's challenge more concrete! Can we always speed up this brute-force algorithm?

1.3.3 The $\mathcal{P}$ versus $\mathcal{NP}$ Question, Its Meaning and Importance

The class $\mathcal{NP}$ is extremely rich (we shall see examples a little later). There are literally thousands of $\mathcal{NP}$ problems in mathematics, optimization, artificial intelligence, biology, physics, economics, industry and other applications that arise naturally out of different necessities and whose efficient solutions will benefit us in numerous ways. They beg for efficient algorithms, but decades of effort (and sometimes more) has succeeded for only a few. Is it possible that *all* sets in $\mathcal{NP}$ possess efficient algorithms, and these simply were not discovered yet? This is the celebrated $\mathcal{P}$ vs. $\mathcal{NP}$ question, which appeared explicitly first in the aforementioned papers of Cook and Levin and was foreshadowed by Gödel.

Open Problem 1.3.4. Is $\mathcal{P} = \mathcal{NP}$?

What explains the abundance of so many natural, important problems in the class $\mathcal{NP}$? Probing the intuitive meaning of the definition of $\mathcal{NP}$, we see that it captures many tasks of human endeavor *for which a successful completion can be easily recognized*. Consider the following professions and the typical tasks they are facing (this will be extremely superficial, but nevertheless instructive):

- **Mathematician:** Given a mathematical claim, come up with a proof for it.
- **Scientist:** Given a collection of data on some phenomena, find a theory explaining it.
- **Engineer:** Given a set of constraints (on cost, physical laws, etc.) come up with a design (of an engine, bridge, laptop, ...) that meets these constraints.
- **Detective:** Given the crime scene, find "who done it."

What is common to all these tasks is that we can typically tell a good solution when we see one (or we at least think we can). In various cases "we" may be the academic community, the customers, or the jury, but we expect the solution to be *short* and *efficiently verifiable*, just as in the definition of $\mathcal{NP}$.

The richness of $\mathcal{NP}$ follows from the simple fact that such tasks abound, and their mathematical formulation is indeed an $\mathcal{NP}$-problem. For all these tasks, efficiency is paramount, and so the importance of the $\mathcal{P}$ vs. $\mathcal{NP}$ problem is evident. The colossal implications of the possibility that $\mathcal{P} = \mathcal{NP}$ are evident as well: *every* instance of these tasks can be solved, optimally and efficiently.

One (psychological) reason people feel that $\mathcal{P} = \mathcal{NP}$ is unlikely is that tasks such as those described above often seem to require a degree of *creativity* that we do not expect a simple computer program to have. We admire Wiles' proof of Fermat's last theorem, the scientific theories of Newton, Einstein, Darwin,

and Watson and Crick, the design of the Golden Gate bridge and the Pyramids, and sometimes even Hercule Poirot's and Miss Marple's analysis of a murder, precisely because they seem to require a leap that cannot be made by everyone, let alone by a simple mechanical device. My own view is that when we finally understand the algorithmic processes of the brain, we may indeed be able to automate the discovery of these specific achievements, and perhaps many others. But can we automate them *all*? Is it possible that for *every* task for which verification is easy, finding a solution is not much harder? If $\mathcal{P} = \mathcal{NP}$, the answer is positive, and creativity (of this abundant, verifiable kind) can be completely automated. Most computer scientists believe that this is not the case.

Conjecture 1.3.5. $\mathcal{P} \neq \mathcal{NP}$.

Back to mathematics! Given the discussion above, one may wonder why it is so hard to prove that indeed $\mathcal{P} \neq \mathcal{NP}$—it seems completely obvious. We shall discuss attempts and difficulties soon, developing a methodology that will enable us to identify the *hardest* problems in $\mathcal{NP}$. But before that, we turn to discuss a related question with a strong relation to mathematics: the $\mathcal{NP}$ versus co$\mathcal{NP}$ question.

1.3.4 The $\mathcal{NP}$ versus co$\mathcal{NP}$ Question, Its Meaning and Importance

Fix a property $C \subseteq \mathrm{I}$. We already have the interpretations

- $C \in \mathcal{P}$ if it is easy to check that object x has property C,
- $C \in \mathcal{NP}$ if it is easy to certify that object x has property C,

to which we now add

- $C \in$ co$\mathcal{NP}$ if it is easy to certify that object x *does not have* property C,

where we formally define:

Definition 1.3.6 (**The class** co$\mathcal{NP}$)**.** A set C is in the class co$\mathcal{NP}$ if and only if its complement $\bar{C} = \mathrm{I} \setminus C$ is in $\mathcal{NP}$.

While the definition of the class $\mathcal{P}$ is symmetric,[13] the definition of the class $\mathcal{NP}$ is asymmetric. Having nice certificates that a given object has property C does not automatically entail having nice certificates that a given object does not have it.

Indeed, when we can do both, we are achieving a mathematics' holy grail of understanding structure, namely *necessary and sufficient* conditions, sometimes phrased as a *duality theorem*. As we know well, such results are rare. When we insist (as we shall do) that the given certificates are *short, efficiently verifiable* ones, they are even rarer. This leads to the conjecture:

[13] Having a fast algorithm to determine if an object has a property C is obviously equivalent to having a fast algorithm for the complementary property $\bar{C}$.

Conjecture 1.3.7. $\mathcal{NP} \neq \text{co}\mathcal{NP}$.

First note that if $\mathcal{P} = \mathcal{NP}$ then $\mathcal{P} = \text{co}\mathcal{NP}$ as well, and hence this conjecture above implies $\mathcal{P} \neq \mathcal{NP}$. We shall discuss at length refinements of this conjecture in Section 1.5 on proof complexity.

Despite the shortage of such efficient complete characterizations, namely properties that are simultaneously in $\mathcal{NP} \cap \text{co}\mathcal{NP}$, they nontrivially exist. Here is a list of some exemplary ones.

- **Linear programming.** Systems of consistent linear inequalities.[14]
- **Zero-sum games.**[15] Finite zero-sum games in which one player can gain at least (some given value) v.
- **Graph connectivity.** The set of graphs in which *every* pair of vertices is connected by (a given number) k disjoint paths.
- **Partial order width.** Finite partial orders whose largest anti-chain has at most (a given number) w elements.
- **Primes.** Prime numbers.

These examples of problems in $\mathcal{NP} \cap \text{co}\mathcal{NP}$ were chosen to make a point. At the time of their discovery (by Farkas, von Neumann, Menger, Dilworth, and Pratt, respectively) the mathematicians working on them were seemingly interested only in characterizing these structures. It is not known if they attempted to find efficient algorithms for these problems. However all of these problems turned out to be in $\mathcal{P}$, with some solutions entering the pantheon of efficient algorithms—for example: the Ellipsoid method of Khachian [38] and the Interior-Point method of Karmarkar [35], both for linear programming, and the recent breakthrough of Agrawal, Kayal, and Saxena [3] for Primes.[16]

Is there a moral to this story? Only that sometimes, when we have an efficient characterization of structure, we can hope for more—efficient algorithms. And conversely, a natural stepping stone toward an elusive efficient algorithm may be to first get an efficient characterization.

Can we expect this magic to always happen? Is $\mathcal{NP} \cap \text{co}\mathcal{NP} = \mathcal{P}$? There are several natural problems in $\mathcal{NP} \cap \text{co}\mathcal{NP}$ that have resisted efficient algorithms for decades, and for some (e.g., factoring integers, computing discrete logarithms) humanity literally banks on their difficulty for electronic commerce security. Indeed, the following is generally believed:

Conjecture 1.3.8. $\mathcal{NP} \cap \text{co}\mathcal{NP} \neq \mathcal{P}$.

Note again that conjecture 1.3.8 implies $\mathcal{P} \neq \mathcal{NP}$, but that it is independent of conjecture 1.3.7.

We now return to develop the main mechanism that will help us study such questions: efficient reductions.

[14] Indeed this generalizes to other convex bodies given by more general constraints, like *semi-definite* programming.

[15] This problem was later discovered to be equivalent to linear programming.

[16] It is interesting that a simple polynomial-time algorithm, whose correctness and efficiency rely on the (unproven) extended Riemann hypothesis, was given thirty years earlier by Miller [42].

1.3.5 Reductions—A Partial Order of Computational Difficulty

In this section, I deal with relating the computational difficulty of problems for which we have no efficient solutions (yet).

Recall that we can regard any classification problem (on finitely described objects) as a subset of our set of inputs I. Efficient reductions provide a natural partial order on such problems that capture their relative difficulty.

Definition 1.3.9 (**Efficient reductions**)**.** Let $C, D \subset \mathrm{I}$ be two classification problems. $f : \mathrm{I} \to \mathrm{I}$ is an efficient reduction from C to D if $f \in \mathcal{P}$ and for every $x \in \mathrm{I}$ we have $x \in C$ if and only if $f(x) \in D$. In this case we call f an *efficient reduction* from C to D. We write $C \leq D$ if there is an efficient reduction from C to D.

The definition of efficient computation allows two immediate observations on the usefulness of efficient reductions. First, that indeed $\leq$ is transitive, and thus defines a partial order. Second, that if $C \leq D$ and $D \in \mathcal{P}$ then also $C \in \mathcal{P}$.

Intuitively, $C \leq D$ means that solving the classification problem C is *computationally* not much harder than solving D. In some cases one can replace *computationally* by the (vague) term *mathematically*. Often such usefulness in mathematical understanding requires more properties of the reduction f than merely being efficiently computable (e.g., we may want it to be represented as a linear transformation or a low dimension polynomial map), and indeed in some cases this is possible. When such a connection between two classification problems (which look unrelated) can be proved, it can mean the portability of techniques from one area to another.

The power of efficient reductions to relate "seemingly unrelated" notions will unfold in later sections. We shall see that they can relate not only classification problems, but such diverse concepts as hardness to randomness; average-case to worst case difficulty; proof length to computation time; the relative power of geometric, algebraic, and logical proof systems; and last but not least, the security of electronic transactions to the difficulty of factoring integers. In a sense, *efficient reductions are the backbone of computational complexity*. Indeed, given that polynomial-time reductions can do all these wonders, it may not be surprising that we have a hard time characterizing the class $\mathcal{P}$!

1.3.6 Completeness

We now return to classification problems. The partial order of their difficulty, provided by efficient reductions, allows us to define the *hardest* problems in a given class. Let $\mathcal{C}$ be any collection of classification problems (namely every element of $\mathcal{C}$ is a subset of I). Of course, here we shall mainly care about the class $\mathcal{C} = \mathcal{NP}$.

Definition 1.3.10 (**Hardness and completeness**)**.** A problem D is called $\mathcal{C}$-*hard* if for every $C \in \mathcal{C}$ we have $C \leq D$. If we further have that $D \in \mathcal{C}$ then D is called $\mathcal{C}$-*complete*.

In other words, if D is $\mathcal{C}$-complete, it is a hardest problem in the class $\mathcal{C}$: if we manage to solve D efficiently, we have done so for all other problems in $\mathcal{C}$. It is not a priori clear that a given class has any complete problems! On the other hand, a given class may have many complete problems, and by definition, they all have essentially the same complexity. If we manage to prove that *any* of them cannot be efficiently solved, then we automatically have done so for *all* of them.

It is trivial, and uninteresting, that every problem in the class $\mathcal{P}$ is in fact $\mathcal{P}$-complete under our definition. It becomes interesting when we find such universal problems in classes of problems for which we do not have efficient algorithms. By far, the most important of all such classes is $\mathcal{NP}$.

1.3.7 $\mathcal{NP}$-completeness

As mentioned earlier, the seminal papers of Cook [16] and Levin [40] defined $\mathcal{NP}$, efficient reducibilities, and completeness, but the crown of their achievement was the discovery of a *natural* $\mathcal{NP}$-complete problem.

Definition 1.3.11 (**The problem *SAT***)**.** A Boolean formula is a logical expression over Boolean variables (that can take values in $\{0,1\}$) with connectives $\wedge, \vee, \neg$, for example: $(x_1 \vee x_2) \wedge (\neg x_3)$. Let SAT denote the set of all satisfiable Boolean formulas (namely those formulas for which there is a Boolean assignment to the variables that gives it the value 1).

Theorem 1.3.12 ([16, 40])**.** *SAT is $\mathcal{NP}$-complete.*

It is a simple exercise to show that Gödel's original problem, as stated in his letter, is $\mathcal{NP}$-complete as well. Moreover, he clearly understood, at least intuitively, its universal nature, captured formally by $\mathcal{NP}$-completeness, a concept discovered fifteen years later.

We recall again the meaning of this theorem. For *every* set $C \in \mathcal{NP}$ there is an efficient reduction $f\colon \mathrm{I} \to \mathrm{I}$ such that $x \in C$ if and only if the formula $f(x)$ is satisfiable! Furthermore, the proof gives an extra bonus which turns out to be extremely useful: given any witness y that $x \in C$ (via some verifier V_C), the same reduction converts the witness y to a Boolean assignment satisfying the formula $f(x)$. In other words, this reduction translates not only between the decision problems, but also between the associated search problems.

You might (justly) wonder how one can prove a theorem like that. Certainly the proof cannot afford to look at all problems $C \in \mathcal{NP}$ separately. The gist of the proof is a generic transformation, taking a description of the verifier V_C for C and emulating its computation on input x and hypothetical witness y to create a Boolean formula $f(x)$ (whose variables are the bits of y). This formula simply tests the validity of the computation of V_C on (x, y) and that this computation outputs 1. Here the locality of algorithms (say, described as Turing machines) plays a central role, as checking the consistency of each step of the computation of V_C amounts simply to a constant size formula on a few bits. To summarize, SAT captures the difficulty of the whole class $\mathcal{NP}$. In particular, the $\mathcal{P}$ vs. $\mathcal{NP}$ problem can now be phrased as a question about the complexity of *one* problem, instead of infinitely many.

Corollary 1.3.13. *$\mathcal{P} = \mathcal{NP}$ if and only if $SAT \in \mathcal{P}$.*

A great advantage of having one complete problem at hand (like SAT) is that now, to prove that another problem (say $D \in \mathcal{NP}$) is $\mathcal{NP}$-complete, we only need to design a reduction from SAT to D (namely prove $SAT \leq D$). We already know that for *every* $C \in \mathcal{NP}$ we have $C \leq SAT$, and transitivity of $\leq$ takes care of the rest.

This idea was used powerfully in Karp's seminal paper [36]. In his paper, he listed twenty-one problems from logic, graph theory, scheduling, and geometry that are $\mathcal{NP}$-complete. This was the first demonstration of the wide spectrum of $\mathcal{NP}$-complete problems and initiated an industry of finding more. A few years later, Garey and Johnson [24] published their book on $\mathcal{NP}$-completeness, which contains hundreds of such problems from diverse branches of science, engineering, and mathematics. Today, thousands are known, in a remarkably diverse set of scientific disciplines.

1.3.8 The Nature and Impact of $\mathcal{NP}$-Completeness

It is hard to do justice to this notion in a couple of paragraphs, but I shall try. More can be found in, for example, [45].

$\mathcal{NP}$-completeness is a unique scientific discovery—there seems to be no parallel scientific notion that pervaded so many fields of science and technology. It became a standard for hardness for problems whose difficulty we have yet no means of proving. It has been used both technically and allegorically to illustrate a difficulty or failure to understand natural objects and phenomena. Consequently, it has been used as a justification for channeling efforts to less ambitious (but more productive) directions. We elaborate below on this effect within mathematics.

$\mathcal{NP}$-completeness has been an extremely flexible and extensible notion, allowing numerous variants that enabled capturing universality in other (mainly computational, but not only) contexts. It led to the ability of defining whole classes of problems by single, universal ones, with the benefits mentioned above. Much of the whole evolution of computational complexity, the theory of algorithms, and most other areas in theoretical computer science has been guided by the powerful approach of reduction and completeness.

It would be extremely interesting to explain the ubiquity of $\mathcal{NP}$-completeness. Being highly speculative for a moment, we can make the following analogies of this mystery with similar mysteries in physics. The existence of $\mathcal{NP}$-completeness in such diverse fields of inquiry may be likened to the existence of the same building blocks of matter in remote galaxies, begging for a common explanation of the same nature as the Big Bang theory. On the other hand, the near lack of natural objects in the (seemingly huge) void of problems in $\mathcal{NP}$ that are neither in $\mathcal{P}$ nor $\mathcal{NP}$-complete raises questions about possible "dark matter," which we have not developed the means of observing yet.

1.3.9 Some $\mathcal{NP}$-complete Problems in Mathematics

Again, I note that all $\mathcal{NP}$-complete problems are equivalent in a very strong sense. Any algorithm solving one can be simply translated into an equally efficient algorithm solving any other. Conversely, if one is difficult then all of them are. I'll explore the meaning of these insights for a number of mathematical classification problems in diverse areas that are NP-complete.

Why did Gödel's incompleteness theorem seemingly have such small effect on working mathematicians? A common explanation is that the unprovable and undecidable problems are far too general compared to those actively being studied. As we shall see below, this argument will not apply to the $\mathcal{NP}$-complete problems we discuss. Indeed, many of these $\mathcal{NP}$-completeness results were proven by mathematicians!

We exemplify this point with two classification problems: $2DIO$, of quadratic Diophantine equations, and $KNOT$, of knots on three-dimensional manifolds.

2DIO: Consider the set of all equations of the form $Ax^2 + By + C = 0$ with integer coefficients A, B, C. Given such a triple, does the corresponding polynomial have a positive integer root (x, y)? Let $2DIO$ denote the subset of triples for which the answer is "yes." Note that this is a very restricted subproblem of the undecidable Hilbert's 10th problem, problem (2) from the Introduction, indeed simpler than even elliptic curves. Nevertheless:

Theorem 1.3.14 ([1]). *The set $2DIO$ is $\mathcal{NP}$-complete.*

KNOT: Consider the set of all triples (M, K, G), representing[17] respectively a three-dimensional manifold M, a knot K embedded on it, and an integer G. Given a triple (M, K, G), does the surface that K bounds have genus at most G? Let $KNOT$ denote the subset for which the answer is "yes."

Theorem 1.3.15 ([2]). *The set $KNOT$ is $\mathcal{NP}$-complete.*

Recall that to prove $\mathcal{NP}$-completeness of a set, one has to prove two things: that it is in $\mathcal{NP}$ and that it is $\mathcal{NP}$-hard. In almost all $\mathcal{NP}$-complete problems, membership in $\mathcal{NP}$ (namely the existence of short certificates) is easy to prove. For example, for $2DIO$ one can easily see that if there is a positive integer solution r to the equation $Ax^2 + By + C = 0$, then indeed there is one whose length (in bits) is polynomial in the lengths of A, B, C, and given such r it is easy to verify that it is indeed a root of the equation. In short, it is very easy to see that $2DIO \in \mathcal{NP}$. But $KNOT$ is an exception, and proving $KNOT \in \mathcal{NP}$ is highly nontrivial. The short witnesses that a given knot has a small genus requires Haken's algorithmic theory of normal surfaces, considerably enhanced (even short certificates for unknottedness in $\mathbb{R}^3$ are hard to obtain, see [33]).

Let us discuss what these $\mathcal{NP}$-completeness results mean, first about the relationship between the two and then about each individually.

The two theorems above and the meaning of $\mathcal{NP}$-completeness together imply that there are *simple* translations (in both directions) between solving

[17] A finite representation can describe M by a triangulation (finite collection of tetrahedra and their adjacencies), and the knot K will be described as a link (closed path) along edges of the given tetrahedra.

$2DIO$ and problem $KNOT$. More precisely, it provides efficiently computable functions $f, h\colon \mathrm{I} \to \mathrm{I}$ performing these translations:

$(A, B, C) \in 2DIO$ if and only if $f(A, B, C) \in KNOT$,

and

$(M, K, G) \in KNOT$ if and only if $h(M, K, G) \in 2DIO$.

So, if we have gained enough understanding of topology to solve, for example, the knot genus problem, it means that we automatically have gained enough number theoretic understanding for solving these quadratic Diophantine problems (and vice versa).

The proofs that these problems are complete both follow by reductions from (variants of) SAT. The combinatorial nature of these reductions may cast doubt on the possibility that the computational equivalence of these two problems implies the ability of real "technology transfer" between topology and number theory. Nevertheless, now that we know of the equivalence, perhaps simpler and more direct reductions can be found between these problems. Moreover, we stress again that for any instance, say $(M, K, G) \in KNOT$, if we translate it using this reduction to an instance $(A, B, C) \in 2DIO$ and happen (either by sheer luck or special structure of that equation) to find an integer root, the same reduction will translate that root back to a description of a genus G manifold that bounds the knot K. Today, many such $\mathcal{NP}$-complete problems are known throughout mathematics, and for some pairs the equivalence can be mathematically meaningful and useful (as it is between some pairs of computational problems).

But regardless of the meaning of the connection between these two problems, there is no doubt what their individual $\mathcal{NP}$-completeness means. Both are mathematically "nasty," as both embed in them the full power of $\mathcal{NP}$. If $\mathcal{P} \neq \mathcal{NP}$, there are no efficient algorithms to describe the objects at hand. Moreover, assuming the stronger $\mathcal{NP} \neq \mathrm{co}\mathcal{NP}$, we should not even expect complete characterization (e.g., above we should not expect short certificates that a given quadratic equation *does not* have a positive integer root).

In short, $\mathcal{NP}$-completeness suggests that we lower our expectations of fully understanding these properties and study perhaps important special cases, variants, etc. Note that such reaction of mathematicians may anyway follow the frustration of unsuccessful attempts at general understanding. However, the stamp of $\mathcal{NP}$-completeness *may* serve as moral justification for this reaction. I stress the word *may*, as the judges for accepting such a stamp can only be the mathematicians working on the problem and how well the associated $\mathcal{NP}$-completeness result captures the structure they try to reveal. I merely point out the usefulness of a formal stamp of difficulty (as opposed to a general feeling) and its algorithmic meaning.

The two examples above come from number theory and topology. I list below some more $\mathcal{NP}$-complete problems from algebra, geometry, optimization, and graph theory. There are numerous other such problems. This will hopefully demonstrate how wide this *modern Gödel phenomena* is in mathematics. The discussion above is relevant to them all.

- **Quadratic equations.** Given a system of multivariate polynomial equations of degree at most 2, over a *finite field* (say GF(2)), do they have a common root?
- **Knapsack.** Given a sequence of integers $a_1, \ldots, a_n$ and b, decide if there exists a subset J such that $\sum_{i \in J} a_i = b$.
- **Integer programming.** Given a polytope in $\mathbb{R}^n$ (by its bounding hyperplanes), does it contain an integer point?
- **Shortest lattice vector.** Given a lattice L in $\mathbb{R}^n$ and an integer k, is the shortest nonzero vector of L of (Euclidean) length $\leq k$?
- **3Color.** Given a graph, can its vertices be colored from {Red, Green, Blue} with no adjacent vertices having the same color?
- **Clique.** Given a graph and an integer k, are there k vertices with all pairs mutually adjacent?

1.4 Lower Bounds, and Attacks on $\mathcal{P}$ versus $\mathcal{NP}$

To prove that $\mathcal{P} \neq \mathcal{NP}$, we must show that for a given problem in $\mathcal{NP}$, no efficient algorithm exists. A result of this type is called a *lower bound* (limiting from below the computational complexity of the problem). Several powerful techniques for proving lower bounds have emerged in the past decades. They apply in two (very different) settings. Below, I describe both and try to explain our understanding of why they seem to stop short of proving $\mathcal{P} \neq \mathcal{NP}$. I mention the first, diagonalization, only very briefly and concentrate on the second, Boolean circuits.

1.4.1 Diagonalization and Relativization

The diagonalization technique goes back to Cantor and his argument that there are more real numbers than algebraic numbers. It was used by Gödel in his incompleteness theorem, and by Turing in his undecidability results, and then refined to prove computational complexity lower bounds. A typical theorem in this area is that more time buys more computational power—for example, there are functions computable in time n^3, say, that are not computable in time n^2. The heart of such arguments is the existence of a "universal algorithm," which can simulate every other algorithm with only small loss in efficiency.

Can such arguments be used to separate $\mathcal{P}$ from $\mathcal{NP}$? This depends on what we mean by "such arguments." The paper by Baker, Gill, and Solovay [6] suggests a formal definition, by exhibiting a feature shared by many similar complexity results, called *relativization*. For example, the result mentioned above separating Turing machines running in time n^3 from those with runnning time n^2 would work perfectly well of all machines in questions we supplied with an "oracle" for any fixed function f, which would answer queries of the form x by $f(x)$ in unit time. So, relativizing lower bounds hold in a "relativized

world" in which any fixed such f is an easy function for the algorithms we consider, and so should hold in *all* such worlds. The [6] paper then proceeded to show that relativizing arguments do not suffice to resolve the $\mathcal{P}$ vs. $\mathcal{NP}$ question. This is done by showing that equipping the machines with different functions f give different answers to the $\mathcal{P}$ vs. $\mathcal{NP}$ question in the respective "relativized worlds" In the three decades since that paper, complexity theory grew far more sophisticated, but nevertheless almost all new results obtained do relativize (one of the few exceptions is in [65]). More on this subject can be found in Chapter 14.3 in [43], Chapter 9.2 of [61], and even more in [22].

1.4.2 Boolean Circuits

A Boolean circuit may be viewed as the "hardware analog" of an algorithm (software). Computation on the binary input sequence proceeds by a sequence of Boolean operations (called *gates*) from the set $\{\wedge, \vee, \neg\}$ (logical AND, OR, and NEGATION) to compute the output(s). We assume that $\wedge, \vee$ are applied to two arguments. We note that while an algorithm can handle inputs of any length, a circuit can handle only one input length (the number of input "wires" it has). A circuit is commonly represented as a (directed, acyclic) graph, with the assignments of gates to its internal vertices. We note that a Boolean formula is simply a circuit whose graph structure is a tree.

Recall that I denotes the set of all binary sequences, and that I_k is the set of sequences of length exactly k. If a circuit has n inputs and m outputs, it is clear that it computes a function $f\colon \mathrm{I}_n \to \mathrm{I}_m$. The efficiency of a circuit is measured by its *size*, which is the analog of time in algorithms.

Definition 1.4.1 (**Circuit size**)**.** Denote by $\mathsf{S}(f)$ the size of the smallest Boolean circuit computing f.

As we care about asymptotic behavior, we shall be interested in sequences of functions $f = \{f_n\}$, where f_n is a function on n input bits. We shall study the complexity $\mathsf{S}(f_n)$ asymptotically as a function of n, and denote it $\mathsf{S}(f)$. For example, let PAR be the parity function, computing if the number of 1's in a binary string is even or odd. Then PAR_n is its restriction to n-bit inputs, and $\mathsf{S}(PAR) = O(n)$.

It is not hard to see that an algorithm (say a Turing machine) for a function f that runs in time T gives rise to a circuit family for the functions f_n of sizes (respectively) $(T(n))^2$, and so efficiency is preserved when moving from algorithms to circuits. Thus proving lower bounds for circuits implies lower bounds for algorithms, and we can try to attack the $\mathcal{P}$ vs. $\mathcal{NP}$ question this way.

Definition 1.4.2 (**The class** $\mathcal{P}$/poly)**.** Let $\mathcal{P}$/poly denote the set of all functions computable by a family of polynomial-size circuits.

Conjecture 1.4.3. $\mathcal{NP} \not\subseteq \mathcal{P}$/poly.

Is this a reasonable conjecture? As mentioned above, $\mathcal{P} \subseteq \mathcal{P}$/poly. Does the reverse inclusion hold? It actually fails badly! There exist undecidable

functions f (which cannot be computed by Turing machines at all, regardless of their running time) that have linear-size circuits. This extra power comes from the fact that circuits for different input lengths share no common description (and thus this model is sometimes called "non-uniform").

So one might expect that proving circuit lower bounds is a much harder task than proving $\mathcal{P} \neq \mathcal{NP}$. However, there is a strong sentiment that the extra power provided by non-uniformity is irrelevant to $\mathcal{P}$ vs. $\mathcal{NP}$. This sentiment comes from a result of Karp and Lipton [37], proving that $\mathcal{NP} \subseteq \mathcal{P}/\text{poly}$ implies a surprising uniform "collapse," not quite $\mathcal{NP} = \text{co}\mathcal{NP}$, but another (somewhat similar but weaker) unlikely collapse of complexity classes.

Still, what motivates replacing the Turing machine by the stronger circuit model when seeking lower bounds? The hope is that focusing on a *finite* model will allow the use of combinatorial techniques to analyze the power and limitations of efficient algorithms. This hope has been realized in the study of interesting, though restricted classes of circuits! The resulting lower bounds for such restricted circuits fall short of resolving the big questions above, but nevertheless have had important applications to computational learning theory, pseudorandomness, proof complexity, and more!

Basic Results and Questions

We have already mentioned several basic facts about Boolean circuits, in particular the fact that they can efficiently simulate Turing machines. The next basic fact, first observed by Shannon [59], is that *most Boolean functions require exponential size circuits.*

This lower bound follows from the gap between the number of functions and the number of small circuits. Fix the number of inputs bits n. The number of possible functions on n bits is precisely 2^{2^n}. On the other hand, the number of circuits of size s is (via a crude estimate of the number of graphs of that size) at most 2^{s^2}. Since every circuit computes one function, we must have $s >> 2^{n/3}$ for *most* functions.

Theorem 1.4.4 ([59])**.** *For almost every function* $f\colon \mathrm{I}_n \to \{0,1\}$, $\mathsf{S}(f) \geq 2^{n/3}$.

Therefore hard functions for circuits (and hence for Turing machines) abound. However, the hardness above is proved via a counting argument, and thus supplies no way of putting a finger on one hard function. I shall return to the nonconstructive nature of this problem in Section 1.5. So far, we cannot prove such hardness for any *explicit* function f (e.g., for an $\mathcal{NP}$-complete function like SAT).

Conjecture 1.4.5. $\mathsf{S}(SAT) = 2^{\Omega(n)}$.

The situation is even worse: no *nontrivial* lower bound is known for any explicit function.[18] Note that for any function f on n bits (which depends on

[18]The notion "explicit," which we repeatedly use here, is a bit elusive and context-dependent at times. But in all cases we seek natural functions, usually of independent mathematical, scientific, or technological interest, rather than functions whose existence is shown by some counting or simulation arguments.

all its inputs), we trivially must have $\mathsf{S}(f) \geq n$, just to read the inputs. The main open problem of circuit complexity is beating this trivial bound.

Open Problem 1.4.6. Find an explicit function $f : \mathrm{I}_n \to \mathrm{I}_n$ for which $\mathsf{S}(f) \neq O(n)$.

Here natural explicit candidate functions may be the multiplication of two $(n/2)$-bit integers, or of two $\sqrt{n} \times \sqrt{n}$ matrices over $GF(2)$. But for any function in $\mathcal{NP}$, such a lower bound would be a breakthrough. Unable to prove any nontrivial lower bound, we now turn to restricted models. There has been some remarkable successes in developing techniques for proving strong lower bounds for natural restricted classes of circuits. I discuss in some detail only one such model.

The 1980s saw a flurry of new techniques for proving circuit lower bounds on natural, restricted classes of circuits. Razborov developed the *Approximation Method*, which allowed proving exponential circuit lower bounds for *monotone* circuits, for such natural problems as *CLIQUE*. The *Random Restriction* method, initiated by Furst, Saxe, and Sipser [23] and Ajtai [4] was used to prove exponential lower bounds on constant depth circuits, for such natural problems as *PARITY*. The *Communication Complexity* method of Karchmer and Wigderson [34] was used to prove lower bounds on monotone formulas—for example, for the *PERFECT MATCHING* problem. See the survey [9] for these and more. But they all fall short of obtaining any nontrivial lower bounds for general circuits, and in particular of proving that $\mathcal{P} \neq \mathcal{NP}$.

Why Is It Hard to Prove Circuit Lower Bounds?

Is there a fundamental reason for this failure? The same may be asked about any longstanding mathematical problem (e.g., the Riemann hypothesis). A natural (vague!) answer would be that, probably, the current arsenal of tools and ideas (which may well have been successful at attacking related, easier problems) does not suffice.

Remarkably, complexity theory can make this vague statement into a theorem! Thus we have a "formal excuse" for our failure so far: we can classify a general set of ideas and tools, which are responsible for virtually all restricted lower bounds known, yet must necessarily fail for proving general ones. This introspective result, developed by Razborov and Rudich [55], suggests a framework called *Natural Proofs*. Very briefly, a lower bound proof is *natural* if it applies to a *large, easily recognizable* set of functions. They first show that this framework encapsulates *all known* lower bounds. Then they show that natural proofs of general circuit lower bounds are unlikely, in the following sense: any natural proof of a lower bound surprisingly implies (as a bi-product) subexponential algorithms for inverting *every* candidate one-way function. Again I stress this irony—natural lower bounds lead to efficient algorithms for the type of problems we want to prove hard!

Specifically, a *natural* (in this formal sense) lower bound would imply subexponential algorithms for such functions as Integer Factoring and Discrete Logarithm, generally believed to be difficult (to the extent that the security of

electronic commerce worldwide relies on such assumptions). This connection strongly uses *pseudorandomness*, which will be discussed later. A simple corollary is that no natural proof exists to show that integer factoring requires circuits of size $2^{n^{1/100}}$ (the best current upper bound is $2^{n^{1/3}}$).

One interpretation of the aforementioned result is an "independence result" of general circuit lower bounds from a certain natural fragment of Peano arithmetic. This may suggest that the $\mathcal{P}$ vs. $\mathcal{NP}$ problem may be independent from Peano arithmetic, or even set theory, which is certainly a possibility.

One final note: it has been over ten years since the publication of the Natural Proof paper. The challenge it raised: *prove a non-natural lower bound*, has not yet been met!

1.5 Proof Complexity

Gödel's letter focuses on lengths of proofs. This section highlights some of the research developments within complexity theory in the last couple of decades on the many facets of this issue. For extensive surveys and different "takes" on this material and its relation to proof theory, independence results, and Gödel's letter, see [7, 11, 46, 47, 48, 50, 57].

The concept of *proof* is what distinguishes the study of mathematics from all other fields of human inquiry. Mathematicians have gathered millennia of experience to attribute such adjectives to proofs as "insightful," "original," "deep," and most notably, "difficult." Can one quantify, mathematically, the difficulty of proving various theorems? This is exactly the task undertaken in proof complexity. It seeks to classify theorems according to the difficulty of proving them, much like circuit complexity seeks to classify functions according to the difficulty of computing them. In proofs, just like in computation, there will be a number of models, called *proof systems*, capturing the power of reasoning allowed to the prover.

Proof systems abound in all areas of mathematics (and not just in logic). Let us see some examples:

1. Hilbert's Nullstellensatz is a (sound and complete) proof system in which *theorems* are inconsistent sets of polynomial equations. A *proof* expresses the constant 1 as a linear combination of the given polynomials.

2. Each finitely presented group can be viewed as a proof system, in which *theorems* are words that reduce to the identity element. A *proof* is the sequence of substituting relations to generate the identity.

3. Reidemeister moves are a proof system in which *theorems* are trivial, unknotted, knots. A *proof* is the sequences of moves reducing the given plane diagram of the knot into one with no crossings.

4. von Neumann's Minimax theorem gives a proof system for every zero-sum game. A *theorem* is an optimal strategy for White, and its *proof* is a strategy for Black with the same value.

In these and many other examples, the *length* of the proof plays a key role, and the quality of the proof system is often related to how short the proofs are that it can provide.

1. In the Nullstellensatz (over fields of characteristic 0), length (of the "coefficient" polynomials, measured usually by their degree and height) usually plays a crucial role in the efficiency of commutative algebra software (e.g., Gröbner basis algorithms).

2. The word problem in general is undecidable. For hyperbolic groups, Gromov's polynomial upper bound on proof length has many uses, of which perhaps the most recent is in his own construction of finitely presented groups with no uniform embedding into Hilbert space [28].

3. Reidemeister moves are convenient combinatorially, but the best upper bounds on length of such proofs (namely on the number of moves) that a given knot is unknotted, are exponential in the description of the knot [32]. Whether one can improve this upper bound to a polynomial is an open question. We note that stronger proof systems were developed to give polynomial upper bounds for proving unknottedness [33].

4. In zero-sum games, happily all proofs are of linear size.

I stress that the asymptotic viewpoint—considering *families* of "theorems" and measuring their proof length as a function of the description length of the theorems—is natural and prevalent. As for computation, this asymptotic viewpoint reveals structure of the underlying mathematical objects, and economy (or efficiency) of proof length often means a better understanding. While this viewpoint is appropriate for a large chunk of mathematical work, you may object that it cannot help explain the difficulty of *single* problems, such as the Riemann hypothesis or $\mathcal{P}$ vs. $\mathcal{NP}$. This is of course a valid complaint. We note however that even such theorems (or conjectures) may be viewed asymptotically (though not always illuminating them better). The Riemann hypothesis has equivalent formulations as a sequence of finite statements, (e.g., about cancelations in the Möbius function). More interestingly, we shall see later a formulation of the $\mathcal{P}$/poly vs. $\mathcal{NP}$ problem as a sequence of finite statements that are strongly related to the Natural Proofs paradigm mentioned above.

All theorems that will concern us in this section are *universal* statements (e.g., an inconsistent set of polynomial equations is the statement that *every* assignment to the variables fails to satisfy them). A short proof for a universal statement constitutes an equivalent formulation that is *existential*—the existence of the proof itself (e.g., the existence of the "coefficient" polynomials in Nullstellensatz that implies this inconsistency). The mathematical motivation for this focus is clear: the ability to describe a property both universally and existentially constitutes *necessary and sufficient* conditions—the afore-mentioned holy grail of mathematical understanding. Here we shall be picky and quantify that understanding according to our usual computational yardstick: the *length* of the existential certificate.

We shall restrict ourselves to *propositional* tautologies. This will automatically give an exponential (thus a known, finite) upper bound on the proof length and will restrict the ballpark (as with $\mathcal{P}$ vs. $\mathcal{NP}$) to the range between polynomial and exponential. The type of statements, theorems, and proofs we shall deal with is best illustrated by the following example.

1.5.1 The Pigeonhole Principle—A Motivating Example

Consider the well-known "pigeonhole principle," stating that there is no injective mapping from a finite set to a smaller one. While trivial, this principle was essential for the counting argument proving the *existence* of exponentially hard functions (Theorem 1.4.4)—this partially explains our interest in its proof complexity. More generally, this principle epitomizes *non-constructive* arguments in mathematics, such as Minkowski's theorem that a centrally symmetric convex body of sufficient volume must contain a lattice point. In both results, the proof does not provide any information about the object proved to exist. Other natural tautologies capture the combinatorial essence of topological proofs (e.g., Brauer's fixed point theorem, the Borsuk–Ulam theorem, Nash's equilibrium)—see [44] for more.

Let us formulate it and discuss the complexity of proving it. First, we turn it into a sequence of finite statements. Fix $m > n$. Let PHP_n^m stand for the statement *there is no one-to-one mapping of m pigeons to n holes*. To formulate it mathematically, imagine an $m \times n$ matrix of Boolean variables x_{ij} describing a hypothetical mapping (with the interpretation that $x_{ij} = 1$ means that the ith pigeon is mapped to the jth hole).[19]

Definition 1.5.1 (**The pigeonhole principle**)**.** The pigeonhole principle PHP_n^m now states that

- either pigeon i is not mapped anywhere (namely, *all* x_{ij} for a fixed i are zeros),
- or some two are mapped to the same hole (namely, for some different i, i', and some j we have $x_{ij} = x_{i'j} = 1$).

These conditions are easily expressible as a formula in the variables x_{ij} (called a *propositional formula*), and the pigeonhole principle is the statement that this formula is a *tautology* (namely satisfied by *every* truth assignment to the variables).

Even more conveniently, the negation of this tautology (which is a *contradiction*) can be captured by a collection of constraints on these Boolean variables that are mutually contradictory. These constraints can easily be written in different languages:

- **Algebraic:** as a set of constant degree polynomials over GF(2).

[19] Note that we do not rule out the possibility that some pigeon is mapped to more than one hole—this condition can be added, but the truth of the principle remains valid without it.

- **Geometric:** as a set of linear inequalities with integer coefficients (to which we seek a $\{0,1\}$ solution).
- **Logical:** as a set of Boolean formulas.

We shall see soon that each setting naturally suggests (several) reasoning tools, such as variants of the Nullstellensatz in the algebraic setting, of Frege systems in the logical setting, and Integer Programming heuristics in the geometric setting. All of these can be formalized as proof systems that suffice to prove this (and any other) tautology. Our main concern will be in the efficiency of each of these proof systems and their relative power, measured in *proof length*. Before turning to some of these specific systems, we discuss this concept in full generality.

1.5.2 Propositional Proof Systems and $\mathcal{NP}$ versus co$\mathcal{NP}$

Most definitions and results in this section come from the paper by Cook and Reckhow [17] that initiated this research direction. I define proof systems and the complexity measure of proof length for each and then relate these to complexity questions we have already met.

All theorems we shall consider will be propositional tautologies. Here are the salient features that we expect[20] from any proof system:

- **Completeness.** Every true statement has a proof.
- **Soundness.** No false statement has a proof.
- **Verification efficiency.** Given a mathematical statement T and a purported proof π for it, it can be easily checked if indeed π proves T in the system. Note that here efficiency of the verification procedure refers to its running-time measured in terms of the *total length of the alleged theorem and proof*.

Remark 1.5.2. Note that we dropped the requirement used in the definition of $\mathcal{NP}$, limiting the proof to be short (polynomial in the length of the claim). The reason is, of course, that proof length is our measure of complexity.

All these conditions are concisely captured, for propositional statements, by the following definition.

Definition 1.5.3 (**Proof systems [17]**)**.** A (*propositional*) *proof system* is a polynomial-time Turing machine M with the property that T is a tautology if and only if there exists a ("*proof*") π such that $M(\pi, T) = 1$.[21]

As a simple example, consider the following "Truth-Table" proof system M_{TT}. Basically, this machine will declare a formula T a theorem if evaluating

[20] Actually, even the first two requirements are too much to expect from strong proof systems, as Gödel famously proved in his incompleteness theorem. However, for propositional statements that have finite proofs, there are such systems.

[21] In agreement with standard formalisms (see below), the proof is seen as coming before the theorem.

it on every possible input makes T true. A bit more formally, for any formula T on n variables, the machine M_{TT} accepts (π, T) if π is a list of *all* binary strings of length n, and for each such string σ, $T(\sigma) = 1$.

Note that M_{TT} runs in polynomial time in its input length, which is the combined length of formula and proof. But in the system M_{TT} proofs are (typically) of exponential length in the size of the given formula. This leads us to the definition of the efficiency (or complexity) of a general propositional proof system M—how short is the shortest proof of each tautology.

Definition 1.5.4 (**Proof length [17]**)**.** For each tautology T, let $\mathsf{S}_M(T)$ denote the size of the shortest proof of T in M (i.e., the length of the shortest string π such that M accepts (π, T)). Let $\mathsf{S}_M(n)$ denote the maximum of $\mathsf{S}_M(T)$ over all tautologies T of length n. Finally, we call the proof system M *polynomially bounded* if and only if for all n we have $\mathsf{S}_M(n) = n^{O(1)}$.

Is there a polynomially bounded proof system (namely one that has polynomial-size proofs for all tautologies)? The following theorem provides a basic connection of this question with computational complexity and the major question of Section 1.3.4. Its proof follows quite straightforwardly from the $\mathcal{NP}$-completeness of SAT, the problem of satisfying propositional formulas (and the fact that a formula is unsatisfiable if and only if its negation is a tautology).

Theorem 1.5.5 ([17])**.** *There exists a polynomially bounded proof system if and only if* $\mathcal{NP} = \mathrm{co}\mathcal{NP}$.

In the next section I focus on natural restricted proof systems. A notion of reduction between proof systems, called *polynomial simulation*, was introduced in [17] and allows us to create a partial order of the relative power of some systems. This is but one example of the usefulness of the methodology developed within complexity theory after the success of $\mathcal{NP}$-completeness.

1.5.3 Concrete Proof Systems

All proof systems in this section are familiar to every mathematician, ever since *The Elements* of Euclid, who formulated a deductive system for plane geometry. In all deductive systems, one starts with a list of formulas, and using simple (and sound!) derivation rules, infers new ones (each formula is called a *line* in the proof).

Normally the initial formulas are taken to be (self-evident) *axioms*, and the final formula derived is the desired theorem. However, here it will be useful to reverse this view and consider *refutation systems*. In the refutation systems below, I start with a contradictory set of formulas and derive a basic contradiction (e.g., $\neg x \wedge x$, $1 = 0$, $1 < 0$), depending on the setting. This serves as proof of the theorem that the initial formulas are mutually inconsistent. I highlight some results and open problems on the proof length of basic tautologies in algebraic, geometric, and logical systems.

Algebraic Proof Systems

We restrict ourselves to the field GF(2). Here a natural representation of a Boolean contradiction is a set of polynomials with no common root. We always add to such a collection the polynomials $x^2 - x$ (for all variables x) that ensure Boolean values (and so we can imagine that we are working over the algebraic closure).

Hilbert's Nullstellensatz suggests a proof system. If $f_1, f_2, \ldots, f_n$ (with any number of variables) have no common root, there must exist polynomials $g_1, g_2, \ldots, g_n$ such that $\sum_i f_i g_i \equiv 1$. The g_i's constitute a proof, and we may ask how short its description is.

A related, but far more efficient system (intuitively based on computations of Gröbner bases) is Polynomial Calculus, abbreviated PC, which was introduced in [14]. The *lines* in this system are polynomials (represented explicitly by all coefficients), and it has two *deduction rules*, capturing the definition of an *ideal*: For any two polynomials g, h and variable x_i, we can use g, h to derive $g + h$, and we can use g and x_i to derive $x_i g$. It is not hard to see (using linear algebra) that if this system has a proof of length s for some tautology, then this proof can be found in time polynomial in s.

Recalling our discussion on $\mathcal{P}$ vs. $\mathcal{NP}$, we do not expect such a property from really strong proof systems.

The PC is known to be exponentially stronger than Nullstellensatz. More precisely, there are tautologies that require exponential length Nullstellensatz proofs, but only polynomial PC proofs. However, strong size lower bounds (obtained from degree lower bounds) are known for PC systems as well. Indeed, the pigeonhole principle is hard for this system. For its natural encoding as a contradictory set of quadratic polynomials, Razborov [53] proved:

Theorem 1.5.6 ([53])**.** *For every n and every $m > n$, $\mathsf{S}_{\mathrm{PC}}(PHP_n^m) \geq 2^{n/2}$, over every field.*

Geometric Proof Systems

Yet another natural way to represent Boolean contradictions is by a set of regions in space containing no integer points. A wide source of interesting contradictions are Integer Programs from combinatorial optimization. Here, the constraints are (affine) linear inequalities with integer coefficients (so the regions are subsets of the Boolean cube carved out by halfspaces). A proof system infers new inequalities from old ones in a way that does not eliminate integer points.

The most basic system is called Cutting Planes (CP), introduced by Chvátal [12]. Its *lines* are linear inequalities with integer coefficients. Its *deduction rules* are (the obvious) addition of inequalities and the (less obvious) dividing the coefficients by a constant (and rounding, taking advantage of the integrality of the solution space).[22]

[22] For example, from the inequality $2x + 4y \geq 1$ we may infer $x + 2y \geq \frac{1}{2}$, and by integrality, $x + 2y \geq 1$.

Let us look at the pigeonhole principle PHP_n^m again. It is easy to express it as a set of contradictory linear inequalities: For every pigeon, the sum of its variables should be *at least* 1. For every hole, the sum of its variables should be *at most* 1. Thus adding up all variables in these two ways implies $m \leq n$, a contradiction. Thus, the pigeonhole principle has polynomial-size CP proofs.

While PHP_n^m is easy in this system, exponential lower bounds were proved for other tautologies. Consider the tautology $CLIQUE_n^k$: No graph on n nodes can simultaneously have a k-clique and a legal $k-1$-coloring. It is easy to formulate it as a propositional formula. Notice that it somehow encodes *many* instances of the pigeonhole principle, one for every k-subset of the vertices.

Theorem 1.5.7 ([49]). $\mathsf{S}_{\mathsf{CP}}\left(CLIQUE_n^{\sqrt{n}}\right) \geq 2^{n^{1/10}}$.

The proof of this theorem by Pudlak [49] is quite remarkable. It *reduces* this proof complexity lower bound to a circuit complexity lower bound. In other words, he shows that any short CP proof of tautologies of certain structure yields a small circuit computing a related Boolean function. You probably guessed that for the tautology at hand the function is indeed the *CLIQUE* function mentioned earlier. Moreover, the circuits obtained are *monotone*, but of the following, very strong form. Rather than allowing only $\wedge, \vee$ as basic gates, they allow *any* monotone binary operation on real numbers! Pudlak then goes to generalize Razborov's approximation method for such circuits and proves an exponential lower bound on the size they require to compute *CLIQUE*.

Logical Proof Systems

The proof systems in this section will all have *lines* that are Boolean formulas, and the differences between them will be in the structural limits imposed on these formulas. We introduce the most important ones: Frege, capturing "polynomial-time reasoning," and Resolution, the most useful system used in automated theorem provers.

The most basic proof system, called the Frege system, puts no restriction on the formulas manipulated by the proof. It has one *derivation rule*, called the *cut rule*: from the two formulas $A \vee C$, $B \vee \neg C$ we may infer the formula $A \vee B$. Every basic book in logic has a slightly different way of describing the Frege system—one convenient outcome of the computational approach, especially the notion of efficient reductions between proof systems, is a proof (in [17]) that they are *all* equivalent, in the sense that the shortest proofs (up to polynomial factors) are independent of which variant you pick!

The Frege system can polynomially simulate *both* the PC and the CP systems. In particular, the counting proof described above for the pigeonhole principle can be carried out efficiently in the Frege system (not quite trivially!), yielding:

Theorem 1.5.8 ([10]). $\mathsf{S}_{\mathsf{Frege}}(PHP_n^{n+1}) = n^{O(1)}$.

Frege systems are basic in the sense that they are the most common in logic and in the sense that polynomial-length proofs in these systems naturally correspond to "polynomial-time reasoning" about feasible objects. In short, this is the proof analog of the computational class $\mathcal{P}$. The major open problem

in proof complexity is to find any tautology (as usual we mean a family of tautologies) that has no polynomial-size proof in the Frege system.

Open Problem 1.5.9. Prove superpolynomial lower bounds for the Frege system.

As lower bounds for Frege are hard, we turn to subsystems of Frege that are interesting and natural. The most widely studied system is Resolution. Its importance stems from its use by most propositional (as well as first-order) *automated theorem provers*, often called Davis–Putnam or DLL procedures [19]. This family of algorithms is designed to find proofs of Boolean tautologies, arising in diverse applications from testing computer chips or communication protocols to basic number theory results.

The *lines* in Resolution refutations are *clauses*, namely disjunctions of literals (like $x_1 \vee x_2 \vee \neg x_3$). The *inference cut rule* simplifies to the *resolution rule*: for two clauses A, B and variable x, we can use $A \vee x$ and $B \vee \neg x$ to derive the clause $A \vee B$.

Historically, the first major result of proof complexity was Haken's[23] [29] exponential lower bound on Resolution proofs for the pigeonhole principle.

Theorem 1.5.10 ([29])**.** $S_{\mathsf{Resolution}}(PHP_n^{n+1}) = 2^{\Omega(n)}$.

To prove it, Haken developed the *bottleneck method*, which is related to both the random restriction and approximation methods mentioned in the circuit complexity chapter. This lower bound was extended to *random tautologies* (under a natural distribution) in [13]. The *width method* of [8] provides much simpler proofs for both results.

1.5.4 Proof Complexity versus Circuit Complexity

These two areas look like very different beasts, despite the syntactic similarity between the local evolution of computation and proof. To begin with, the number of objects they care about differs drastically. There are doubly exponentially number of functions (on n bits), but only exponentially many tautologies of length n. Thus a counting argument shows that some functions (albeit nonexplicit) require exponential circuit lower bounds (Theorem 1.4.4), but no similar argument can exist to show that some tautologies require exponential size proofs. So while we prefer lower bounds for natural, explicit tautologies, *existence* results of hard tautologies for strong systems are interesting in this setting as well.

Despite the different nature of the two areas, there are deep connections between them. Quite a few of the techniques used in circuit complexity, most notably *Random Restrictions*, were useful for proof complexity as well. The lower bound we saw in the previous section is extremely intriguing: a monotone circuit lower bound directly implies a (nonmonotone) proof system lower bound! This particular type of reduction, known as the *Interpolation Method*, was successfully used for other, weak proof systems, like Resolution. This leads to the

[23] Armin Haken, the son of Wolfgang Haken, cited earlier for his work on knots.

question: can reductions of a similar nature be used to obtain lower bounds for a strong system (like Frege) from (yet unproven) circuit lower bounds?

Open Problem 1.5.11. Does $\mathcal{NP} \not\subseteq \mathcal{P}/\text{poly}$ imply superpolynomial Frege lower bounds?

Why are Frege lower bounds hard? The truth is, we do not know. The Frege system (and its relative, Extended Frege), capture *polynomial-time reasoning*, as the basic objects appearing in the proof are polynomial-time computable. Thus superpolynomial lower bounds for these systems is the proof complexity analog of proving superpolynomial lower bounds in circuit complexity. As we saw, for circuits we at least understand to some extent the limits of existing techniques, via Natural Proofs. However, there is no known analog of this framework for proof complexity.

I conclude with a tautology capturing the $\mathcal{P}/\text{poly}$ vs. $\mathcal{NP}$ question, thus using proof complexity to try to show that proving circuit lower bounds is difficult.

This tautology, suggested by Razborov, simply encodes the statement $\mathcal{NP} \not\subseteq \mathcal{P}/\text{poly}$, namely that SAT does not have small circuits. More precisely, fix n, an input size to SAT, and s, the circuit size lower bound we attempt to prove.[24] The variables of our "Lower Bound" formula LB_n^s encode a circuit C of size s, and the formula simply checks that C disagrees with SAT on at least one instance ϕ of length n (namely that either $\phi \in SAT$ and $C(\phi) = 0$ or $\phi \notin SAT$ and $C(\phi) = 1$). Note that LB_n^s has size $N = 2^{O(n)}$, so we seek a superpolynomial in N lower bound on its proof length.[25]

Proving that LB_n^s is hard for Frege will in some sense give another explanation to the difficulty of proving circuit lower bound. Such a result would be analogous to the one provided by Natural Proofs, only without relying on the existence of *one-way* functions. But paradoxically, the same inability to prove circuit lower bounds seems to prevent us from proving this proof complexity lower bound! Even proving that LB_n^s is hard for Resolution has been extremely difficult. It involves proving hardness of a *weak* pigeonhole principle[26]—one with exponentially more pigeons than holes. It was finally achieved with the tour-de-force of Raz [52], and further strengthening of [54].

Acknowledgements

I am grateful to Scott Aaronson and Christos Papadimitriou for carefully reading and commenting on an earlier version this manuscript. I acknowledge support from NSF grant CCR-0324906. Some parts of this paper are revisions of material taken from my ICM 2006 paper "$\mathcal{P}, \mathcal{NP}$ and Mathematics: A Computational Complexity View."

[24] For example, we may choose $s = n^{\log \log n}$ for a superpolynomial bound, or $s = 2^{n/1000}$ for an exponential one.

[25] Of course, if $\mathcal{NP} \subseteq \mathcal{P}/\text{poly}$ then this formula is *not* a tautology, and there is no proof at all.

[26] This explicates the connection mentioned between the pigeonhole principle and the counting argument proving existence of hard functions.

Bibliography

[1] L. Adleman and K. Manders. Computational complexity of decision problems for polynomials. *Proceedings of 16th IEEE Symposium on Foundations of Computer Science*. Los Alamitos (CA): IEEE Comput. Soc. Press, 1975, pp. 169–77.

[2] I. Agol, J. Hass, and W. P. Thurston. The computational complexity of knot genus and spanning area. *Trans. Amer. Math. Sci.* **358** (2006), 3821–50.

[3] M. Agrawal, N. Kayal, and N. Saxena. Primes is in $\mathcal{P}$. *Ann. of Math.* **160** (2) (2004), 781–93.

[4] M. Ajtai. Σ_1-formulae on finite structures. *Ann. Pure Appl. Logic* **24** (1) (1983), 1–48.

[5] S. Arora and B. Barak. *Computational Complexity: A Modern Approach.* New York: Cambridge University Press, 2009.

[6] T. Baker, J. Gill, and R. Solovay. Relativizations of the P = ?NP question. *SIAM J. Comput.* **4** (1975), 431–42.

[7] P. Beame and T. Pitassi. Propositional proof complexity: past, present, and future. *Bull. EATCS* **65** (1998), 66–89.

[8] E. Ben-Sasson and A. Wigderson. Short proofs are narrow—resolution made simple. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing*. New York: ACM Press, 1999, pp. 517–26.

[9] R. Boppana and M. Sipser The complexity of finite functions. In *Handbook of Theoretical Computer Science, Volume A, Algorithms and Complexity*, ed. J. van Leeuwen. Amsterdam: Elsevier Science Publishers, B.V.; Cambridge (MA): MIT Press, 1990, pp. 757–804.

[10] S. Buss. Polynomial size proofs of the propositional pigeonhole principle. *J. Symbolic Logic* **52** (1987), 916–27.

[11] S. Buss. On Gödel's theorems on lengths of proofs II: Lower bounds for recognizing k symbol provability. In *Feasible Mathematics II*, ed. P. Clote and J. Remmel. Boston: Birkhauser, 1995, pp. 57–90.

[12] V. Chvátal. Edmonds polytopes and a hierarchy of combinatorial problems. *Discrete Math.* **4** (1973), 305–37.

[13] V. Chvátal and E. Szemerédi. Many hard examples for resolution. *J. ACM* **35** (4) (1988), 759–68.

[14] M. Clegg, J. Edmonds, and R. Impagliazzo. Using the Groebner basis algorithm to find proofs of unsatisfiability. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing*. New York: ACM Press, 1996, pp. 174–83.

[15] A. Cobham. The intrinsic computational difficulty of functions. In *Logic, Methodology, and Philosophy of Science*. North Holland, Amsterdam, 1965, pp. 24–30.

[16] S. A. Cook. The complexity of theorem-proving procedures. *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*. New York: ACM Press, 1971, pp. 151–8.

[17] S. A. Cook and R. A. Reckhow. The relative efficiency of propositional proof systems. *J. Symbolic Logic* **44** (1979), 36–50.

[18] T. H. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. 2nd edn. Cambridge (MA): MIT Press; New York: McGraw-Hill Book Co., 2001.

[19] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *J. ACM* **5** (7) (1962), 394–7.

[20] J. Edmonds. Minimum partition of a matroid into independent sets. *Journal of Research of the National Bureau of Standards (B)*, **69** (1965), 67–72.

[21] J. Edmonds. Paths, trees, and flowers. *Canad. J. Math.* **17** (1965), 449–67.

[22] L. Fortnow. The role of relativization in complexity theory. *Bull. EATCS* **52** (1994), 229–44.

[23] M. Furst, J. Saxe, and M. Sipser. Parity, circuits and the polynomial time hierarchy. *Math. Systems Theory* **17** (1984), 13–27.

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

[25] O. Goldreich *Computational Complexity: A Conceptual Perspective*. New York: Cambridge University Press, 2008.

[26] A. Granville. It is easy to determine whether a given integer is prime. *Bull. Amer. Math. Soc.* **42** (2005), 3–38.

[27] M. Gromov. Hyperbolic groups. In *Essays in Group Theory*, ed. S. M. Gersten, Math. Sci. Res. Inst. Publ. 8. New York: Springer-Verlag, 1987, pp. 75–264.

[28] M. Gromov. Random walk in random groups. *Geom. Funct. Anal.* **13** (1) (2003), 73–146.

[29] A. Haken. The intractability of resolution. *Theor. Comput. Sci.* **39** (1985), 297–308.

[30] W. Haken. Theorie der Normalflächen: ein Isotopiekriterium für den Kreisknoten. *Acta Math.* **105** (1961), 245–375.

[31] J. Hartmanis. Gödel, von Neumann and the P = ?NP problem. *Bull. EATCS* **38** (1989), 101–7.

[32] J. Hass and J. C. Lagarias. The number of Reidemeister moves needed for unknotting. *J. Amer. Math. Soc.* **14** (2001), 399–428.

[33] J. Hass, J. C. Lagarias, and N. Pippenger. The computational complexity of knot and link problems. *J. ACM* **46** (1999), 185–211.

[34] M. Karchmer and A. Wigderson, Monotone circuits for connectivity require super-logarithmic depth. *SIAM J. Discrete Math.* **3** (2) (1990), 255–65.

[35] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica* **4** (1984), 373–94.

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

[37] R. Karp and R. J. Lipton. Turing machines that take advice. *Enseign. Math.* (2) **28** (1982), 191–209

[38] L. Khachian. A polynomial time algorithm for linear programming. *Soviet Math. Doklady* **10** (1979), 191–4.

[39] A. K. Lenstra, H. W. Lenstra Jr., and L. Lovász. Factoring polynomials with rational coefficients. *Math. Ann.* **261** (1982), 515–34.

[40] L. A. Levin. Universal search problems. *Probl. Peredaci Inform.* **9** (1973), 115–6; English transl. *Probl. Inf. Transm.* **9** (1973), 265–6.

[41] Y. V. Matiasevich. *Hilbert's Tenth Problem.* Cambridge (MA): MIT Press, 1993.

[42] G. L. Miller. Riemann's hypothesis and tests for primality. *J. Comput. System Sci.* **13** (3) (1976), 300–17.

[43] C. H. Papadimitriou. *Computational Complexity.* Reading (MA): Addison Wesley, 1994.

[44] C. H. Papadimitriou. On the complexity of the parity argument and other inefficient proofs of existence. *J. Comput. System Sci.* **48** (3) (1994), 498–532.

[45] C. H. Papadimitriou. NP-completeness: A retrospective. In *Automata, Languages and Programming* (ICALP 1997), Lecture Notes in Computer Science, vol. 1256, ed. P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela. Berlin: Springer-Verlag, 1997, pp. 2-6.

[46] P. Pudlak. Logic and complexity: independence results and the complexity of propositional calculus. In *Proceedings of International Congress of Mathematicians 1994*, 3–11 Aug 1994, Zurich, Switzerland, ed. S. D. Chatterji. Basel: Birkhäuser Verlag, 1995, pp. 288–97.

[47] P. Pudlak. A bottom-up approach to foundations of mathematics. In *Gödel '96: Logical Foundations of Mathematics, Computer Science and Physics—Kurt Godel's Legacy* (Proceedings), Lecture Notes in Logic, vol. 6, ed. P. Hajek. Berlin: Springer-Verlag, 1996, pp. 81–97.

[48] P. Pudlak. On the lengths of proofs of consistency. *Collegium Logicum, Annals of the Kurt-Godel-Society* **2** (1996), pp. 65–86.

[49] P. Pudlak. Lower bounds for resolution and cutting planes proofs and monotone computations. *J. Symbolic Logic* **62** (3) (1997), 981–98.

[50] P. Pudlak. The lengths of proofs. In *Handbook of Proof Theory*, ed. S.R. Buss. Amsterdam: Elsevier, 1998, pp. 547–637.

[51] M. Rabin. Mathematical theory of automata. In *Mathematical Aspects of Computer Science*, Proceedings of Symposia in Applied Mathematics, vol. 19. Providence (RI): American Mathematical Society, 1967, pp. 153–75.

[52] R. Raz. Resolution lower bounds for the weak pigeonhole principle. *J. ACM* **51** (2) (2004), 115–38.

[53] A. A. Razborov. Lower bounds for the polynomial calculus. *Comput. Complexity* **7** (4) (1998), 291–324.

[54] A. A. Razborov. Resolution lower bounds for perfect matching principles. *J. Comput. System Sci.* **69** (1) (2004), 3–27.

[55] A. A. Razborov and S. Rudich. Natural proofs. *J. Comput. System Sci.* **55** (1) (1997), 24–35.

[56] N. Robertson and P. Seymour. Graph Minors I–XIII. *J. Combin. Theory B* (1983–1995).

[57] S. Rudich and A. Wigderson (eds.). *Computational Complexity Theory*, IAS/Park City Mathematics Series, vol. 10. American Mathematical Society/Institute for Advanced Studies, 2000.

[58] A. Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*, Algorithms and Combinatorics, vol. 24. Berlin: Springer-Verlag, 2003.

[59] C. E. Shannon. The synthesis of two-terminal switching circuits,. *Bell Systems Technical Journal*, **28** (1949), 59–98.

[60] M. Sipser. The history and status of the P versus NP question. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*. New York: ACM Press, 1992, pp. 603–18.

[61] M. Sipser. *Introduction to the Theory of Computation*. Boston (MA): PWS Publishing Co., 1997.

[62] A. Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, 1951.

[63] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, Ser. 2, **42** (1936), 230–65.

[64] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. A correction. *Proc. London Math. Soc.*, Ser. 2, **43** (1937), 544–6.

[65] N. V. Vinodchandran. $\mathrm{AM}_{\mathrm{exp}} \not\subseteq (\mathrm{NP} \cap \mathrm{coNP})/\mathrm{poly}$. *Inform. Process. Lett.* **89** (2004), 43–7.