Appendix B

On the Quest for Lower Bounds

Alas, Philosophy, Medicine, Law, and unfortunately also Theology, have I studied in detail, and still remained a fool, not a bit wiser than before. Magister and even Doctor am I called, and for a decade am I sick and tired of pulling my pupils by the nose and understanding that we can know nothing.[1]

J.W. Goethe, Faust, Lines 354–364

Summary: In this appendix we survey some attempts at proving lower bounds on the complexity of natural computational problems. In the first part, devoted to Circuit Complexity, we describe lower bounds for the *size* of (restricted) circuits that solve natural computational problems. This can be viewed as a program whose long-term goal is proving that $\mathcal{P} \neq \mathcal{NP}$. In the second part, devoted to Proof Complexity, we describe lower bounds on the length of (restricted) propositional proofs of natural tautologies. This can be viewed as a program whose long-term goal is proving that $\mathcal{NP} \neq \text{co}\mathcal{NP}$.

The current activity in these areas is aimed towards developing proof techniques that may be applied to the resolution of the "big problems" (such as P versus NP), but the current achievements (though very impressive) seem far from reaching this goal. Current crown-jewel achievements in these areas take the form of tight (or strong) lower bounds on the complexity of computing (resp., proving) "relatively simple" functions (resp., claims) in *restricted* models of computation (resp., proof systems).

[1] This quote reflects a common sentiment, not shared by the author of the current book.

B.1 Preliminaries

Circuit complexity refers to a non-uniform model of computation; specifically the model of Boolean circuits, focusing on the size of such circuits, while ignoring the complexity of constructing adequate circuits. Similarly, proof complexity refers to proofs of tautologies, focusing on the length of such proofs, while ignoring the complexity of generating such proofs. Both circuits and proofs are finite objects that are defined on top of the notion of a *directed acyclic graph* (dag), which we review next.

A dag $G(V,E)$ consists of a finite set of vertices V, and a set of ordered pairs called directed edges $E \subseteq V \times V$, in which there are no directed cycles. The vertices with no incoming edges are called the inputs of the dag G, and the vertices with no outgoing edges are called the outputs. We will restrict ourselves to dags in which the number of *incoming* edges to every vertex is at most 2. If the number of *outgoing* edges from every node is at most 1, the dag is called a tree. Finally, we assume that every vertex can be reached from some input via a directed path. The size of a dag will be its number of edges.

To make a dag into a computational device (or a proof), each non-input vertex will be marked by a rule, converting values in its predecessors to values at that vertex. It is easy to see that the vertices of every dag can be linearly ordered, such that predecessors of every vertex (if any) appear before it in the ordering. Thus, if the input vertices are labeled with some values, we can label the remaining vertices (in that order), one at a time, till all vertices (and in particular all outputs) are labeled.

For computation devices, the non-input vertices will be marked by functions (called gates), which make the dag a circuit. If we label the input vertices by specific values from some domain, the outputs will be determined by them, and the circuit will naturally define a function (from input values to output values). For more details see Section 1.2.4.

For proofs, the non-input vertices will be marked by sound deduction (or inference) rules, which make the dag a proof. If we label the inputs by formulae that are axioms in a given proof system, the output again will be determined by them, and will yield the tautology proved by this proof.

We note that both settings fit the paradigm of simplicity shared by all computational models discussed in Section 1.2; the rules are simple by definition – they are applied to at most 2 previous values. The main difference is that this model is finite – each dag can compute only functions/proofs with a fixed input length. To allow all input lengths, one must consider infinite sequences of dags, one for each length, thus obtaining a model of computing devices having infinite description (when referring to all input lengths). This significantly extends the power of the computation model beyond that of the notion of *algorithm* (discussed in Section 1.2.3). However, as we are interested in lower bounds here, this is legitimate, and one can hope that the finiteness of the model will potentially allow for combinatorial techniques to analyze its power and limitations. Furthermore, these models allow for the introduction (and study) of meaningful restricted classes of computations.

B.2 Boolean Circuit Complexity

In Boolean circuits all inputs, outputs, and values at intermediate nodes of the dag are bits. The set of allowed gates is naturally taken to be a *complete basis* – one that allows the circuit to compute *all* Boolean functions. The specific choice of a complete basis hardly effects the study of circuit complexity. A typical choice is the set $\{\wedge, \vee, \neg\}$ of (respectively) conjunction, disjunction (each on 2 bits) and negation (on 1 bit).

For a finite function f, we denote by $\mathcal{S}(f)$ the size of the smallest Boolean circuit computing f. We will be interested in sequences of functions $\{f_n\}$, where f_n is a function on n input bits, and will study (their size complexity) $\mathcal{S}(f_n)$ asymptotically as a function of n. With some abuse of notation, for $f(x) \stackrel{\text{def}}{=} f_{|x|}(x)$, we let $\mathcal{S}(f)$ denote the integer function that assigns to n the value $\mathcal{S}(f_n)$. Thus, we refer to the following definition.

Definition B.1 (circuit complexity): *Let $f : \{0,1\}^* \to \{0,1\}^*$ and $\{f_n\}$ be such that $f(x) = f_{|x|}(x)$ for every x. The* complexity *of a function f* (resp., *$\{f_n\}$*), *denoted $\mathcal{S}(f)$* (resp., denoted *$n \mapsto \mathcal{S}(f_n)$*), *is a function of n that represents the size of the smallest Boolean circuit computing f_n.*

We note that different circuits (e.g., having a different number of inputs) are used for each f_n. Still there may be a simple description of this sequence of circuits, say, an algorithm that on input n produces a circuit computing f_n. In case such an algorithm exists and works in time polynomial in the size of its output, we say that the corresponding sequence of circuits is uniform. Note that if f has a uniform sequence of polynomial-size circuits then $f \in \mathcal{P}$. On the other hand, it can be shown that any $f \in \mathcal{P}$ has (a uniform sequence of) polynomial-size circuits. Consequently, a super-polynomial size lower-bound on any function in $\mathcal{NP}$ would imply that $\mathcal{P} \neq \mathcal{NP}$.

Definition B.1 makes no reference to "uniformity" and indeed the sequence of smallest circuits computing $\{f_n\}$ may be highly "nonuniform". Indeed, non-uniformity makes the circuit model stronger than Turing machines (or, equivalently, than the model of uniform circuits): there exist functions f that cannot be computed by Turing machines (regardless of their running time), but do have linear-size circuits. So isn't proving circuit lower-bounds a much harder task than we need to resolve the P vs. NP question?

The answer is that there is a strong sentiment that the extra power provided by non-uniformity is irrelevant to the P vs. NP question; that is, it is conjectured that NP-complete sets do not have polynomial-size circuits. This conjecture is supported by the fact that its failure will yield an unexpected collapse in the complexity world of standard computations (see Section 3.2). Furthermore, the hope is that abstracting away the (supposedly irrelevant) uniformity condition will allow for combinatorial techniques to analyze the power and limitations of polynomial-size circuits (w.r.t NP-sets). This hope has materialized in the study of restricted classes of circuits (see Sections B.2.2 and B.2.3). Indeed, another advantage of the circuit model is that it offers a framework for naturally restricted models of computation.

We also mention that Boolean circuits are a natural computational model, corresponding to "hardware complexity" (which was indeed the original motivation for their introduction by Shannon [194]), and so their study is of independent interest. Moreover, some of the techniques for analyzing Boolean functions found applications elsewhere (e.g., in computational learning theory, combinatorics and game theory).

B.2.1 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. Another basic fact is that *most Boolean functions require exponential size circuits*, which is due to the gap between the number of functions and the number of small circuits.

Thus, hard functions (i.e., function that require large circuits and thus have no efficient algorithms) do exist, to say the least. However, the aforementioned hardness result is proved via a counting argument, and so provides no way of pointing to one hard function. Using more conventional language, we cannot prove analogous hardness results for any *explicit* function f (e.g., for an NP-complete function like SAT or even for functions in $\mathcal{EXP}$). The situation is even worse: no *nontrivial* lower-bound is known for any explicit function. Note that for any function f on n bits (which depends on all its inputs), we trivially must have $\mathcal{S}(f) \geq n$, just to read the inputs. One major open problem of circuit complexity is beating this trivial bound.

Open Problem B.2 *Find an explicit Boolean function f (or even a length-preserving function f) for which $\mathcal{S}(f)$ is not $O(n)$.*

A particularly basic special case of this problem, is the question whether addition is easier to perform than multiplication. Let $\mathtt{ADD}:\{0,1\}^n\times\{0,1\}^n\to\{0,1\}^{n+1}$ and $\mathtt{MULT}:\{0,1\}^n\times\{0,1\}^n\to\{0,1\}^{2n}$, denote the addition and multiplication functions, respectively, applied to a pair of integers (presented in binary). For addition we have an optimal upper bound; that is, $\mathcal{S}(\mathtt{ADD}) = O(n)$. For multiplication, the standard (elementary school) quadratic-time algorithm can be greatly improved (via Discrete Fourier Transforms) to slightly super-linear, yielding $\mathcal{S}(\mathtt{MULT}) = O(n \cdot (\log n)^2)$. Now, the question is *whether or not there exist linear-size circuits for multiplication* (i.e., is $\mathcal{S}(\mathtt{MULT}) = O(n)$)?

Unable to report on any nontrivial lower-bound (for an explicit function), we turn to restricted models. There has been some remarkable successes in developing techniques for proving strong lower-bounds for natural restricted classes of circuits. We describe the most important ones, and refer the reader to [43, 225] for further detail.

General Boolean circuits, as described above, can compute every function and can do it at least as efficiently as general (uniform) algorithms. Restricted circuits may be only able to compute a subclass of all functions (e.g., monotone functions). The restriction makes sense when the related classes of functions and the computations represented by the restricted circuits are natural (from a conceptual or practical viewpoint). The models discussed below satisfy this condition.

B.2.2 Monotone Circuits

An extremely natural restriction comes by forbidding negation from the set of gates, namely allowing only $\{\wedge, \vee\}$. The resulting circuits are called monotone circuits and it is easy to see that they can compute every function $f : \{0,1\}^n \to \{0,1\}$ that is monotone with respect to the standard partial order on n-bit strings (i.e., $x \preceq y$ iff for every bit position i we have $x_i \leq y_i$). A very natural question in this context is whether or not non-monotone operations (in the circuit) help in computing monotone functions?

Before turning to this question, we note that it is as easy to see that most monotone functions require exponential size circuits (let alone monotone ones).[2] Still, proving a super-polynomial lower-bound on the monotone circuit complexity of an explicit monotone function was open for over 40 years, till the invention of the so-called *approximation method* (by Razborov [180]).

Let `CLIQUE` be the function that, given a graph on n vertices (by its adjacency matrix), outputs 1 if and only if the graph contains a complete subgraph of size (say) $\sqrt{n}$ (i.e.,, all pairs of vertices in some $\sqrt{n}$ subset are connected by edges). This function is clearly monotone. Moreover, it is known to be NP-complete.

Theorem B.3 ([180], improved in [7]): *There are no polynomial-size monotone circuits for* `CLIQUE`.

We note that the lower-bounds are sub-exponential in the number of vertices (i.e., size $\exp(\Omega(n^{1/8}))$ for n vertices), and that similar lower-bounds are known for functions in $\mathcal{P}$. Thus, *there exists an exponential separation between monotone circuit complexity and non-monotone circuit complexity*, where this separation refers (of course) to the computation of monotone functions.

B.2.3 Bounded-Depth Circuits

The next restriction is structural: *we allow all gates, but limit the depth of the circuit*. The depth of a dag is simply the length of the longest directed path in it. So in a sense, depth captures the *parallel time* to compute the function: if a circuit has depth d, then the function can be evaluated by enough processors in d phases (where in each phase many gates are evaluated at once). Indeed, parallel time is a natural and important computational resource, referring to the following basic question: *can one speed up computation by using several computers in parallel?* Determining which computational tasks can be "parallelized" when many processors are available and which are "inherently sequential" is clearly a fundamental question.

We will restrict d to be a constant, which still is interesting not only as a measure of parallel time but also due to the relation of this model to expressibility in first order logic as well as to complexity classes above NP called the *Polynomial-time*

[2] A key observation is that it suffices to consider the set of n-bit monotone functions that evaluate to 1 (resp., to 0) on each string $x = x_1 \cdots x_n$ satisfying $\sum_{i=1}^{n} x_i > \lfloor n/2 \rfloor$ (resp., $\sum_{i=1}^{n} x_i < \lfloor n/2 \rfloor$). Note that each such function is specified by $\binom{n}{\lfloor n/2 \rfloor}$ bits.

Hierarchy (see Section 3.2). In the current setting (of constant-depth circuits), we allow *unbounded fan-in* (i.e., $\wedge$-gates and $\vee$-gates taking any number of incoming edges), as otherwise each output bit can depend only on a constant number of input bits.

Let PAR (for parity) denote the sum modulo two of the input bits, and MAJ (for majority) be 1 if and only if there are more 1's than 0's among the input bits. The invention of the *random restriction method* (by Furst, Saxe, and Sipser [79]) led to the following basic result.

Theorem B.4 ([79], improved in [229, 111]): *For all constant d, the functions* PAR *and* MAJ *have no polynomial size circuit of depth d.*

The aforementioned improvement (of Håstad [111], following Yao [111]) gives a relatively tight lower-bound of $\exp(\Omega(n^{1/(d-1)}))$ on the size of n-input PAR circuits of depth d.

Interestingly, MAJ remains hard (for constant-depth polynomial-size circuits) even if the circuits are also allowed (unbounded fan-in) PAR-gates (this result is based on yet another proof technique: *approximation by polynomials* [201, 181]). However, the "converse" does not hold (i.e., constant-depth polynomial-size circuits with MAJ-gates can compute PAR), and in general the class of constant-depth polynomial-size circuits with MAJ-gates (denoted $\mathcal{TC}^0$) seems quite powerful. In particular, nobody has managed to prove that there are functions in $\mathcal{NP}$ that cannot be computed by such circuits, even if the depth is restricted to 3.

B.2.4 Formula Size

The final restriction is again structural – we require the dag to be a tree. Intuitively, this forbids the computation from reusing a previously computed partial result (and if it is needed again, it has to be recomputed). Thus, the resulting Boolean circuits are simply Boolean formulae. (Indeed, we are back to the basic model allowing negation ($\neg$), and $\wedge, \vee$ gates *of fan-in 2*.)

Formulae are natural not only for their prevalent mathematical use, but also because their size can be related to the depth of general circuits and to the *memory* requirements of Turing machines (i.e., their space complexity). One of the oldest results on Circuit Complexity, is that PAR and MAJ have nontrivial lower-bounds in this model. The proof follows a simple combinatorial (or information theoretic) argument.

Theorem B.5 [138]: *Boolean formulae for n-bit* PAR *and* MAJ *require $\Omega(n^2)$ size.*

This should be contrasted with the linear-size circuits that exist for both functions. We comment that $\mathcal{S}(\text{PAR}) = O(n)$ is trivial, but $\mathcal{S}(\text{MAJ}) = O(n)$ is not. Encouraged by Theorem B.5, one may ask whether we can hope to provide super-polynomial lower-bounds on the formula size of explicit functions. This is indeed a famous open problem.

Open Problem B.6 *Find an explicit Boolean function f for which $\mathcal{S}(f)$ is super-polynomial.*

One of the cleanest methods suggested is the *communication complexity method* (of Karchmer and Wigderson [135]). This method asserts that the depth of a formula for a Boolean function f equals the communication complexity in the following two party game, G_f. The first party is given $x \in f^{-1}(1) \cap \{0,1\}^n$, the second party is given $y \in f^{-1}(0) \cap \{0,1\}^n$, and their goal is to find a bit location on which x and y disagree (i.e., i such that $x_i \neq y_i$, which clearly exists). To that end, the party exchange messages, according to a predetermined protocol, and the question is what is the communication complexity (in terms of total number of bits exchanged on the worst-case input pair) of the best such protocol.

Note that proving a super-logarithmic lower-bound on the communication complexity of the aforementioned game G_f will establish a super-polynomial lower-bound on the size of formulae computing f (because formula depth can be made logarithmic in their size). We stress that a lower-bound of purely information theoretic nature (no computational restriction were paced on the parties in the game) implies a computational lower-bound!

We mention that the communication complexity method has a *monotone version* in which the depth of monotone circuits is related to the communication complexity of protocols that are required to find an i such that $x_i > y_i$ (rather than any i such that $x_i \neq y_i$).[3] In fact, the monotone version is better known than the general one, due to its success in establishing linear lower-bounds on monotone depth of natural problems such as perfect matching (by Raz and Wigderson [179]).

B.3 Arithmetic Circuits

We now leave the Boolean rind, and discuss circuits over general fields. Fix any field F. The gates of the dag will now be the standard $+$ and $\times$ operations in the field. This requires two immediate clarifications. First, to allow using constants of the field, one adds a special input vertex whose value is the constant '1' of the field. Moreover, multiplication by any field element (e.g., -1) is free. Second, one may wonder about division. However, we will be mainly interested in computing polynomials, and for computing polynomials (over infinite fields) division can be efficiently emulated by the other operations.

Now the inputs of the dag will hold elements of the field F, and hence so will all computed values at vertices. Thus an arithmetic circuit computes a polynomial map $p : F^n \to F^m$, and every such polynomial map is computed by some circuit. We denote by $\mathcal{S}_F(p)$ the size of a smallest circuit computing p (when no subscript is given, $F = \mathcal{Q}$ the field of rational numbers). As usual, we'll be interested in sequences of polynomials, one for every input size, and will study size asymptotically.

It is easy to see that over any *fixed* finite field, arithmetic circuits can simulate Boolean circuits on Boolean inputs with only constant factor loss in size. Thus

[3] Note that since f is monotone, $f(x) = 1$ and $f(y) = 0$ implies the existence of an i such that $x_i = 1$ and $y_i = 0$.

the study of arithmetic circuits focuses more on infinite fields, where lower bounds may be easier to obtain.

As in the Boolean case, the existence of hard functions is easy to establish (via dimension considerations, rather than counting argument), and we will be interested in *explicit* (families of) polynomials. However, the notion of explicitness is more delicate here (e.g., allowing polynomials with algebraically independent coefficients would yield strong lower-bounds, which are of no interest whatsoever). Very roughly speaking, polynomials are called explicit if the mapping from monomials to (a finite description of) their coefficients has an efficient program.

An important parameter, which is absent in the Boolean model, is the *degree* of the polynomial(s) computed. It is obvious, for example, that a degree d polynomial (even in one variable, i.e., $n = 1$) requires size at least $\log d$. We briefly consider the univariate case (where d is the only measure of input size), which already contains striking and important problems. Then we move to the general multivariate case, in which as usual n, the number of inputs will be the main parameter (where we shall assume that $d \leq n$). We refer the reader to [82, 207] for further detail.

B.3.1 Univariate Polynomials

How tight is the $\log d$ lower-bounds for the size of an arithmetic circuit computing a degree d polynomial? A simple dimension argument shows that for most degree d polynomials p, it holds that $\mathcal{S}(p) = \Omega(d)$. However, we know of no explicit one:

Open Problem B.7 *Find an explicit polynomial p of degree d, such that $\mathcal{S}(p)$ is not $O(\log d)$.*

To illustrate the question, we consider the following two concrete polynomials $p_d(x) = x^d$, and $q_d(x) = (x+1)(x+2)\cdots(x+d)$. Clearly, $\mathcal{S}(p_d) \leq 2\log d$ (via repeated squaring), so the trivial lower-bound is essentially tight. On the other hand, it is a major open problem to determine $\mathcal{S}(q_d)$, and the conjecture is that $\mathcal{S}(q_d)$ is not polynomial in $\log d$. To realize the importance of this question, we state the following proposition:

Proposition B.8 *If $\mathcal{S}(q_d) = \mathrm{poly}(\log d)$, then the integer factorization problem can be solved by polynomial-size circuits.*

Recall that it is widely believed that the integer factorization problem is intractable (and, in particular, does not have polynomial-size circuits). Proposition B.8 follows by observing that $q_d(t) \equiv ((t+d)!)/(t!) \pmod N$ and that using a circuit for q_d we can efficiently obtain the value of $((t+d)!)/(t!) \bmod N$ (by emulating the computation of the former circuit modulo N). Furthermore, the value of $(K!) \bmod N$ can be obtained from a product of some of the polynomials q_{2^j} evaluated at adequate points. Next, observe that $(K!) \bmod N$ and N are relatively prime if and only if all prime factors of N are bigger than K. Thus, given a composite N, we can find a factor of N by performing a binary search for a suitable K.

B.3.2 Multivariate Polynomials

We are now back to polynomials with n variables. To make n our only input size parameter, it is convenient to restrict ourselves to polynomials whose total degree is at most n.

Once again, almost every polynomial p in n variables requires size $\mathcal{S}(p) \geq \exp(n/2)$, and we seek explicit polynomial (families) that are hard. Unlike in the Boolean world, here there are slightly nontrivial lower-bounds (via elementary tools from algebraic geometry).

Theorem B.9 [24]: $\mathcal{S}(x_1^n + x_2^n + \cdots + x_n^n) = \Omega(n \log n)$.

The same techniques extend to prove a similar lower-bound for other natural polynomials such as the symmetric polynomials and the determinant. Establishing a stronger lower-bound for any explicit polynomial is a major open problem. Another open problem is obtaining a super-linear lower-bound for a polynomial map of constant (even 1) total degree. Outstanding candidates for the latter open problem are the *linear* maps computing the Discrete Fourier Transform over the Complex numbers, or the Walsh transform over the Rationals (for both $O(n \log n)$ algorithms are known, but no super-linear lower-bounds are known).

We now focus on specific polynomials of central importance. The most natural and well studied candidate for the last open problem is the matrix multiplication function MM: let A, B be two $m \times m$ matrices of variables over F, and define $\mathtt{MM}(A, B)$ to be the $n = m^2$ entries of the matrix $A \times B$. Thus, MM is a set of n explicit bilinear forms over the $2n$ input variables. It is known that $\mathcal{S}_{\mathrm{GF}(2)}(\mathtt{MM}) \geq 3n$ (cf., [198]). On the other hand, the obvious $m^3 = n^{3/2}$ algorithm can be improved.

Theorem B.10 [59]: *For every field F, $\mathcal{S}_F(\mathtt{MM}) = O(n^{1.19})$.*

So what is the complexity of MM (even if one counts only multiplication gates)? Is it linear or almost-linear or is it the case that $\mathcal{S}(\mathtt{MM}) > n^\alpha$ for some $\alpha > 1$? This is indeed a famous open problem.

We next consider the determinant and permanent polynomials (DET and PER, resp.) over the $n = m^2$ variables representing an $m \times m$ matrix. While DET plays a major role in classical mathematics, PER is somewhat esoteric in that context (though it appears in Statistical Mechanics and Quantum Mechanics). In the context of complexity theory both polynomials are of great importance, because they capture natural complexity classes. The function DET has relatively low complexity (and is closely related to the class of polynomials having polynomial-sized arithmetic formulae), whereas PER seems to have high complexity (and it is complete for the counting class $\#\mathcal{P}$ (see §6.2.1)). Thus, it is conjectured that PER is *not* polynomial-time reducible to DET. One restricted type of reduction that makes sense in this algebraic context is a reduction by projection.

Definition B.11 (projections): *Let $p_n : F^n \to F^\ell$ and $q_N : F^N \to F^\ell$ be polynomial maps and $x_1, ..., x_n$ be variables over F. We say that there is a* projection from p to q over F, *if there exists a function $\pi : [N] \to \{x_1, ..., x_n\} \cup F$ such that $p(x_1, ..., x_n) \equiv q(\pi(1), ..., \pi(N))$.*

Clearly, if $p_n \propto q_N$ then $\mathcal{S}_F(p_n) \leq \mathcal{S}_F(q_N)$. Let $\mathtt{DET}_m$ and $\mathtt{PER}_m$ denote the functions DET and PER restricted to m-by-m matrices. It is known that $\mathtt{PER}_m \propto \mathtt{DET}_{3^m}$, but to yield a polynomial-time reduction one would need a projection of $\mathtt{PER}_m$ to $\mathtt{DET}_{\mathrm{poly}(m)}$. It is conjectured that no such projection exists.

B.4 Proof Complexity

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.

We will consider only propositional proof systems, and so our theorems will be *tautologies*. We will see soon why the complexity of proving tautologies is highly nontrivial and amply motivated.

The formal definition of a proof system spells out what we take for granted: the efficiency of the verification procedure. In the following definition the efficiency of the verification procedure refers to its running-time measured in terms of the *total length of the alleged theorem and proof*. In contrast, in Chapter 9, we consider the running-time as a function of the *length of the alleged theorem*. (Both approaches were mentioned in Section 2.1, where the two approaches coincide because in Section 2.1 we mandated proofs of length polynomial in the alleged theorem.)

Definition B.12 [58]: *A* (propositional) proof system *is a polynomial-time Turing machine M such that a formula T is a tautology of and only if exists a string π, called a* proof, *such that $M(\pi, T) = 1$.*

In agreement with standard formalisms (see below), the proof is viewed as coming before the theorem. Note that Definition B.12 guarantees the completeness and soundness of the proof system, as well as verification efficiency (relative to the total length of the alleged proof–theorem pair). Definition B.12 judiciously ignores the length of the proof π (of the tautology T), viewing the length of the proof as a measure of the complexity of the tautology T with respect to the proof system M.

For each tautology T, let $\mathcal{L}_M(T)$ denote the length of the shortest proof of T in M (i.e., the length of the shortest string π such that M accepts (π, T)). That is, $\mathcal{L}_M$ captures the *proof complexity* of various tautologies with respect to the proof system M.

Abusing notation, we let $\mathcal{L}_M(n)$ denotes the maximum $\mathcal{L}_M(T)$ over all tautologies T of length n. The following simple theorem provides a basic connection between proof complexity (with respect to any propositional proof system) and computational complexity (i.e., the NP-vs-coNP Question).

Theorem B.13 [58]: *There exists a propositional proof system M such that $\mathcal{L}_M$ is polynomial if and only if $\mathcal{NP} = \text{co}\mathcal{NP}$.*

In particular, a propositional proof system M such that $\mathcal{L}_M$ is polynomial coincides with a NP-proof system (as in Definition 2.5) for the set of propositional tautologies, which is a co$\mathcal{NP}$-complete set.

The long-term goal of Proof Complexity is to establish super-polynomial lower-bounds on the length of proofs in any propositional proof system (and thus establish $\mathcal{NP} \neq \text{co}\mathcal{NP}$). It is natural to start this formidable project by considering first simple (and thus weaker) proof systems, and then move on to more and more complex ones. Moreover, various natural proof systems, capturing basic (restricted) types and "primitives" of reasoning as well as natural tautologies, suggest themselves as objects for this study. In the rest of this section we focus on such restricted proof systems. Different branches of Mathematics such as logic, algebra and geometry provide different such systems, often implicitly. A typical system would have a set of axioms, and a set of deduction rules. A proof would proceed to derive the desired tautology in a sequence of steps, each producing a formula (often called a line of the proof), which is either an axiom, or follows from previous formulae via one of the deduction rules.

Regarding these proof systems, we make two observations. First, proofs in these systems can be easily verified by an algorithm (and thus they fit the general framework of Definition B.12). Second, these proof systems perfectly fit our dag model. The inputs will be labeled by the axioms, the internal vertices by deduction rules, which in turn "infer" a formula for that vertex from the formulae at the vertices pointing to it.[4]

For various proof systems Π, we turn to study the proof length $\mathcal{L}_\Pi(T)$ of tautologies T in proof system Π. The first observation, revealing a major difference between proof complexity and circuit complexity, is that the trivial counting argument *fails*. The reason is that, while the number of functions on n bits is 2^{2^n}, there are at most 2^n tautologies of this length. Thus, in proof complexity, even the *existence* of a hard tautology, not necessarily an explicit one, would be of interest (and, in particular, if established for all propositional proof systems then it would yield $\mathcal{NP} \neq \text{co}\mathcal{NP}$). (Note that here we refer to hard instances of of a problem and not to hard problems.) Anyhow, as we shall see, most known lower-bounds (in restricted proof systems) apply to very natural (let alone explicit) tautologies.

Conventions: There is an equivalent and somewhat more convenient view of (simple) proof systems, namely as (simple) refutation systems. First, recalling that 3SAT is NP-complete, note that every (negation of a) tautology can be written as a conjunction of clauses, with each clause being a disjunction of only 3 literals (variables or their negation). Now, if we take these clauses as axioms, and derive (using the rules of the system) a contradiction (e.g., the negation of an axiom, or better yet the empty clause), then we have proved the tautology (since we have

[4] General proof systems as in Definition B.12 can also be adapted to this formalism, by considering a deduction rule that corresponds to a single step of the machine M. However, the deduction rules considered below are even simpler, and more importantly they are natural.

proved that its negation yields a contradiction). Proof complexity often takes the refutation viewpoint, and often exchanges "tautology" with its negation ("contradiction").

The rest of this section is divided to three parts, referring to logical, algebraic and geometric proof systems. We will briefly describe important representative and basic results in each of these domains, and refer the reader to [25] for further detail (and, in particular, to adequate references).

B.4.1 Logical Proof Systems

The proof systems in this section will all have lines that are Boolean formulae, and the differences will be in the structural limits imposed on these formulae.

The most basic proof system, called Frege system, puts no restriction on the formulae manipulated by the proof. It has one derivation rule, called the cut rule: $A \vee C, B \vee \neg C \vdash A \vee B$ (adding any other sound rule, like *modus ponens*, has little effect on the length of proofs in this system). Frege systems are basic in the sense that they (in several variants) are the most common in Logic, and in that polynomial length proofs in these systems naturally corresponds to "polynomial-time reasoning" about feasible objects.

The major open problem in proof complexity is to find any tautology (as usual, we mean a family of tautologies) that has no polynomial-long proof in the Frege system.

Since lower-bounds for Frege are hard, we turn to subsystems of Frege which are interesting and natural. The most widely studied system is Resolution, whose importance stems from its use by most propositional (as well as first order) automated theorem provers. The formulae allowed in Resolution refutations are simply clauses (disjunctions), and so the derivation cut rule simplifies to the "resolution rule": $A \vee x, B \vee \neg x \vdash A \vee B$, for clauses A, B and variable x.

An example of a tautology that is easy for Frege and hard for Resolution, is the pigeonhole principle, $\mathtt{PHP}_n^m$, expressing the fact that there is no one-to-one mapping of m pigeons to $n < m$ holes.

Theorem B.14 $\mathcal{L}_{\text{Frege}}(\mathtt{PHP}_n^{n+1}) = n^{O(1)}$ *but* $\mathcal{L}_{\text{Resolution}}(\mathtt{PHP}_n^{n+1}) = 2^{\Omega(n)}$

B.4.2 Algebraic Proof Systems

Just as a natural contradiction in the Boolean setting is an unsatisfiable collection of clauses, a natural contradiction in the algebraic setting is a system of polynomials without a common root. Moreover, CNF formulae can be easily converted to a system of polynomials, one per clause, over any field. One often adds the polynomials $x_i^2 - x_i$ which ensure Boolean values.

A natural proof system (related to Hilbert's Nullstellensatz, and to computations of Grobner bases in symbolic algebra programs) is Polynomial Calculus, abbreviated PC. The lines in this system are polynomials (represented explicitly by all coefficients), and it has two deduction rules: For any two polynomials g, h, the rule $g, h \vdash g + h$, and for any polynomial g and variable x_i, the rule $g, x_i \vdash x_i g$.

Strong length lower-bounds (obtained from degree lower-bounds) are known for this system. For example, encoding the pigeonhole principle as a contradicting set of constant degree polynomials, we have

Theorem B.15 *For every n and every $m > n$, $\mathcal{L}_{\mathrm{PC}}(\mathtt{PHP}_n^m) \geq 2^{n/2}$, over every field.*

B.4.3 Geometric Proof Systems

Yet another natural way to represent contradictions is by a set of regions in space that have empty intersection. Again, we care mainly about discrete (say, Boolean) domains, and 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 half-spaces). The most basic system is called Cutting Planes (CP). Its lines are linear inequalities with integer coefficients. Its deduction rules are (the obvious) addition of inequalities, and the (less obvious) division of the coefficients by a constant (and rounding, taking advantage of the integrality of the solution space).

While $\mathtt{PHP}_n^m$ is easy in this system, exponential lower-bounds are known for other tautologies. We mention that they are obtained from the *monotone circuit* lower bounds of Section B.2.2.