A Status Report on the P versus NP Question

Eric Allender[*]
Department of Computer Science
Rutgers University
New Brunswick, NJ 08855, USA
allender@cs.rutgers.edu

Abstract

We survey some of the history of the most famous open question in computing: the P versus NP question. We summarize some of the progress that has been made to date, and assess the current situation.

Contents

[*]Supported in part by NSF Grants CCF-0830133, CCF-0832787, and DMS-0652582.

1 Prologue

Does the world really need another survey of the P vs NP question? There are excellent textbooks that deal with this topic at length (here is a partial list: [11, 32, 41, 97]) and there have been a number of short survey articles by eminent authors [86, 30, 54, 78, 98] – not to mention the excellent series of NP-completeness columns written by David Johnson [56] that serve as an on-going “status report” on the P vs NP question. What reasons can be given, to justify spending time and effort in creating yet another overview of this topic?

First, I must confess to a selfish motive. I love computational complexity theory, and I think that the world is a better place if more people have the opportunity to learn something about the topic. The *Advances in Computer Science* series has a venerable pedigree and has published wonderful papers about progress on many exciting topics, but there has *not* been an update on the P versus NP problem in this series. I see this is an opportunity to claim some prime real estate and use it for a noble purpose: to trumpet the news that there has been thrilling and spectacular progress on several fronts in complexity theory. We have learned that the world of complexity is, in some ways, a much stranger place than anyone could have suspected back in the early days of the field. (In particular, the theory of probabilistically checkable proofs has led to very counterintuitive conclusions.) Simultaneously, the overarching lesson is that the computational universe exhibits much more structure than we should have had any right to hope for; a surprisingly small collection of tools allows us to categorize the overwhelming majority of computational problems that we really want to understand. How could I toss aside an opportunity to spread the good news about complexity theory?

Second, the editor’s plea was hard to refuse:

> *I am interested in a chapter on complexity theory and the chronology of the ‘P=NP?’ problem – What is it? Why is it important? What has been done over the past 30 years? And what is its current status? Has any progress been made since I was a graduate student? What is the impact if we do solve the question? What would it mean in the long run if P=NP? if P is not equal to NP? if it is proven undecidable if P=NP?*

This is an excellent list of questions. In fact, it provides me with a ready-made outline for this paper. Surely there are many readers of the *Advances in Computer Science* series that share his feelings and questions. How could I turn my back on such a need?

But the third and final reason is this: The P versus NP problem deals with the central mystery of computation. The story of the long assault on this problem is our Iliad and our Odyssey; it is the defining myth of our field. Just as authors throughout history have returned time and again to the classic heroic tales in order to reveal new aspects of the human condition, so do complexity theoreticians take up the task of re-telling the story of the P versus NP question from different perspectives. Most of the main plot developments that will be recounted here have been described quite well in the surveys that I list in the first paragraph of this prologue. I will try not to duplicate the efforts of the authors of those surveys. In particular, I will not give a detailed history of the past thirty years in complexity theory; the reader should consult [98] for an excellent exposition of many of the developments that I discuss here, providing a more detailed bibliography and many interesting insights. I do not claim that this version of the story is superior to any of the recent surveys cited above – but at least it is different from all of them, and brings out certain parts of the story that need to be told. A great story is always worth re-telling.

2 What is the 'P=NP?' Problem?

In the beginning, there was the reduction.

The story of NP-completeness begins with the story of the unreasonable effectiveness of *reducibility* as a tool to characterize the complexity of computational problems that we really care about solving. Prior to the breakthroughs of Cook [28], Levin [64], and Karp [58], many of the fundamental properties of computational complexity had already been worked out. For instance:

- The complexity of a problem f can be measured in terms of the size of the smallest circuit computing f [84] or in terms of the running time of a program computing f [46].
- "Most" functions on n input bits require circuits of nearly maximal (exponential) size [84].
- For "nice" time bounds $t < T$, programs running in time T can accomplish more than programs running in time t [46] (and the restriction to "nice" time bounds is essential [21]).
- It is tricky (but not impossible) to formalize the notion of a "tight lower bound" on the running time required to compute a function [67], because some problems provably have nothing remotely like an "optimal" algorithm [20].

All of this was lovely and important – but something vital was missing. Nobody had any idea how to say anything concrete about the computational complexity of any *natural* problem. (The word "natural" appears over and over again in the literature of computational complexity, although it seems impossible to give an adequate definition of what people mean when they refer to problems being "natural". A good rule of thumb is that a function f is a "natural" computational problem if you can imagine someone being paid to produce a program or a circuit that computes f. The functions f that are shown to be hard to compute via diagonalization arguments [46, 20] fail this test of "naturalness".) Thus the field of computational complexity, which by rights should lie at the heart of our understanding of practical computation, instead was perched perilously close to the no-man's-land of irrelevance.

This is what changed as a result of the work of Cook, Levin, and Karp [28, 64, 58]. Their work introduced an audacious new tool: *efficient reductions among computational problems*. At this point we need to do three things:

1. Explain what an efficient reduction is,
2. Explain why this is an audacious notion, and
3. Explain why this was such a big deal.

2.1 What is an efficient reduction?

Let's focus on the most basic and most useful version of reducibility: An *efficient reduction* is a function (i.e., a transformation that takes a bit string as input and produces a bit string as output) that is "easy to compute". Initially (in the work of Cook, Levin, and Karp) attention centered on *polynomial time reducibility*, meaning that a reduction f was considered "easy" if $f(x)$ could be computed in time at most $p(n)$, where n is the length of the input string x, and p is a polynomial. We'll also consider other notions of "easy" later on.

In order to talk about "reducibility among computational problems" using this notion of "reduction", it is convenient to limit our attention to computational problems that produce a "yes" or "no" answer (such as the problem of taking a graph as input and determining if the graph is connected, or taking (the binary representation of) a number as input and determining if the number is composite). Most computational problems can easily be re-stated in this way. Thus a computational problem can be viewed as a *set*: namely, we can identify a computational problem with the set of input instances x for which the correct output is "yes".

Given two computational problems A and B, we say that A is *efficiently reducible to* B if there is an efficient reduction f such that x is in A if and only if $f(x)$ is in B. That is, input instances of problem A can be "easily" encoded as input instances of problem B; knowing the answer as to whether or not $f(x) \in B$ yields the answer of whether x is in A. Many people find the name "reduction" confusing. The use of this term traces back to the notion of "reducing" fractions (such as re-writing 2/4 as 1/2). My battered edition of Webster's Dictionary [71] characterizes this as a process "to change (an expression) to an equivalent but more fundamental expression". Thus a reduction f showing that A is reducible to B is a way to change an instance of A into an equivalent instance of B; we are "distilling" the computational essence of problem A, and showing that, at heart, it consists of nothing much more complicated than B.

The notion of reducibility is much older than complexity theory. The concept of a mapping on bit strings inducing a "reduction" from one computational problem to another was already firmly established as a tool for showing that certain problems could not be solved by computer programs (regardless of the running time). See any textbook on computability theory (such as [52]) for more background. The work of Cook, Levin, and Karp merely hijacked this well-known notion, and imposed time bounds.

2.2 Why is this an audacious notion?

On the face of things, the notion of polynomial-time reducibility does not seem like a very promising way to forge a link between practical concerns of real-world computing and the abstract theory of computational complexity. According to this definition, a function f that takes inputs of length n and requires $n^{1,000,000}$ computational steps to compute is to be considered "easy to compute", even though the sun would most likely have become extinct long before any computing device would have a chance to compute $f(x)$ for inputs x consisting of even a handful of bits. How can anyone be serious in proposing such a preposterous definition of "easy to compute"?

The reason behind this fraud is quite simple: It is convenient to maintain the fiction that if f and g are easy to compute, then it should also be easy to compute $f(g(x))$. This makes "reducibility" a transitive relation, so that if A is reducible to B and B is reducible to C, then A is also reducible to C. There are many classes of functions that one could use that would have this property. For instance, we could consider a function to be "easy to compute" only if it could be computed in time linear in the input length, or we could restrict attention to running time bounded by $O(n \log^k n)$ for different values k. These notions have been studied, but they have not turned out to be nearly as useful in characterizing "natural" problems as polynomial-time reducibility has.

Focusing on polynomial-time reducibility brings other benefits, too. There are lots of computer programs in wide use that run for time n^2 on inputs of size n. One would not want to exclude such transformations from the class of "easy" functions. But once one allows quadratic time, there is no easy way to avoid allowing arbitrary polynomial time; a function f computable in time n^2 may produce output of size n^2. Thus computing $f(f(x))$ will take time n^4, and the reader can easily see where this leads. If the composition of two "easy" functions is to be considered "easy", then one is led quite quickly to consider all polynomial-time-computable functions as being "easy".

Another benefit of basing complexity theory on polynomial time is that, when we are measuring run-time, we are freed from (almost all) concerns about being specific about the type of computing device that our programs are running on, as well as what language the programs are written in, etc. This is because, in the history of computing thus far, almost all "real-world" notions of computing that have been proposed can easily be simulated with only a polynomial slow-down by Turing machines. This is sometimes referred to as the "Invariance Thesis" [87] or the "Strong Church-Turing Thesis" [19, 29]. (Don't worry if you don't know what Turing machines are; think of them as a particularly simple programming language and machine architecture. Occasionally one hears an objection to the Turing machine model, indicating that Turing machines are *more* powerful than physical computational devices, since they come equipped with an *infinite* memory. However, this objection misses the point. Programs are typically written to handle inputs of *any* length; if a program is run on a machine with very limited memory the program may be unable to execute properly on very large inputs, although the same program would run fine on a machine that has more memory. Turing machines are intended to model *programs*, rather than *machines*; the existence of a fast program for a given problem is equivalent to the existence of a fast Turing machine for the problem. Rather than thinking of Turing machines as being *more* powerful than physical computers, it is more accurate to think of Turing machines as being a very *restricted* class of programs.)

The Strong Church-Turing Thesis is not universally accepted. Probabilistic computation (augmenting computers with a source of random bits [40]) and quantum computing (see the work of Shor [85] for insight into the power of this model) have both been proposed as physically-realizable programming paradigms that might provide more computational power. There are good reasons for considering these and other models of computation, and there are good reasons for being skeptical about whether problems computable in polynomial time are truly feasible, but there is no doubt about the fact that the class of problems solvable in polynomial time now occupies a central position in the way that we understand computation.

Definition 1 *The class* P *consists of all computational problems* A *for which there is a polynomial* p *and a Turing machine that takes input* x *of length* n *and determines whether* x *is in* A*, in time bounded by* $p(n)$.

It is certainly not true that all problems in P are easy to compute. The value of the definition lies in the fact that there is good reason to consider problems that do *not* lie in P to be *hard* to compute.

2.3 Why was this such a big deal?

Efficient reducibility provided the first useful abstraction that enabled us to make sense of a chaotic universe of computational problems. Prior to the development of this tool, it was known that some problems had efficient algorithms while no efficient algorithms had been discovered for other problems, but there was no way to estimate the likelihood of an efficient algorithm being found, if none was already known.

In what sense does efficient reducibility provide an abstraction? If A is efficiently reducible to B, and B is efficiently reducible to A, then in a very meaningful sense, A and B are "equivalent" – they are merely two different ways of looking at the same problem. Thus instead of infinitely many different apparently-unrelated computational problems, one can deal instead with a much smaller number of *classes* of equivalent problems. Technically, there are still infinitely-many of these classes; it is known that for any problem A that lies outside P there are infinitely many classes that lie "between" P and A [63, 51] – but these constructions rely on "unnatural" computational problems, which are not the computational problems that people really care about. The amazing fact (which is also amazingly useful) is that "natural" computational problems tend to clump into a just a few equivalence classes. This was completely unexpected. Nothing had prepared the computing community for the shocking insight that there are really just a handful of *fundamentally different* computational problems that people want to solve.

2.4 Complexity Classes

The story just keeps getting better. Not only is there a framework of classes of equivalent problems that helps us partition natural computational problems into meaningful groups, but many of of these equivalence classes correspond in a meaningful way to resource bounds.

Let us illustrate this by means of an example. Consider the problem of computing optimal next moves in a game of checkers. (Checkers is played on an 8-by-8 grid; we actually consider the generalized problem that is played on an n-by-n grid.) Optimal strategies for n-by-n checkers can be computed in time exponential in n, and thus this problem lies in the class known as EXP (for "exponential time"). It turns out that Checkers is also a canonical problem for EXP, in the following sense: every problem $A \in$ EXP is polynomial-time reducible to Checkers [81]. That is, every problem in EXP can be rephrased as a problem of finding an optimal move in a game of checkers. This is a specific case of a very general phenomenon, known as "hardness".

Definition 2 *Let $\mathcal{C}$ be a class of computational problems. A set A is* hard for *$\mathcal{C}$ if B is polynomial-time reducible to A for every set $B \in \mathcal{C}$.*

Definition 3 *Let $\mathcal{C}$ be a class of computational problems. A set A is* complete for *$\mathcal{C}$ if A is hard for $\mathcal{C}$ and $A \in \mathcal{C}$.*

Thus Checkers is complete for EXP. This means that the computational complexity of this problem is fairly well understood; it can be solved in exponential time, and it cannot be solved in time much less than 2^n because

(a) it is known [46] that there is *some* problem A in EXP that requires time $\Omega(2^n)$,

(b) by completeness, A is reducible to Checkers in time n^k for some k, and

(c) if Checkers were solvable in time less than $2^{n^{1/k}}$, the reduction from (b) would yield an algorithm for A that runs too quickly, violating the lower bound (a).

If only things were always this nice! Next, we consider a completeness theorem that produces a much less satisfactory result.

Consider the problem of determining if two regular expressions (i.e., the type of expression that is used in the tool "grep") are equivalent (in the sense that they denote the same regular set). This problem is complete for PSPACE (the class of problems that can be solved by Turing machines using at most $p(n)$ memory locations on inputs of length n, for some polynomial p) [72]. This is quite similar in spirit to the claim that Checkers is complete for EXP, and we can still legitimately claim that determining equivalence of regular expressions is a canonical problem for PSPACE and in some sense is one of the most difficult problems in PSPACE – but there is a significant difference. We do not know if PSPACE is equal to P, and thus we cannot (currently) conclude *anything* about the time that is required of programs that solve this problem.

...But there is more to be learned from this example. Even though we cannot currently *prove* that programs require exponential time to solve the regular expression equivalence problem, we can say that it would be a dramatic breakthrough if any subexponential-time algorithm for this problem were to emerge. It would imply similarly fast algorithms for *every* PSPACE-complete problem (and many such problems are known, *all* of which have resisted fast algorithmic solution), and it would mean that *any* problem that can be solved by an algorithm that uses n^k memory (including algorithms that search through all 2^{n^k} strings in a large search space looking for possible solutions to a problem) can be solved fairly quickly, in time much less than is required to examine a search space of size 2^{n^k}. It seems inconceivable that this should be possible, although we still have no formal proof that it is really impossible. Taken together, this is very strong evidence that the regular expression equivalence problem is hard to compute, even if it falls short of the standard of a real proof.

P, PSPACE, and EXP are three important examples of *complexity classes*: classes of computational problems that consist of all of the problems that can be computed using a certain amount of some computational resource (such as time or memory). Unfortunately, many important natural problems do *not* seem to be complete for any class that is defined in terms of bounding computational resources on realistic models of computation. This motivates turning to *unrealistic* models of computation.

2.5 The class NP

Not only did Cook, Levin, and Karp introduce efficient reducibility as a useful tool for classifying the complexity of natural problems, but they also focused attention on the one complexity class that has turned out to be more useful than any other: NP.

Unlike the complexity classes that have been discussed thus far (P, PSPACE, EXP), NP is not defined in terms of computation on machines that are intended to model real-world computing. A *nondeterministic* machine that runs in time t is provided with access to a "magic word" m of length t, in addition to its ordinary input x. We say that the machine *accepts* input x if there is any word m that could be provided to it, that would cause the computation on input (x, m) to output 1. Otherwise, we say that the machine *rejects* its input x. Note that this is roughly the same thing as allowing a machine to search through all of the 2^t words of length t, looking for a solution that would cause it to output 1. However, we say that the running time of the machine is t, instead of the time 2^t that it would take to search through the entire list of possibilities deterministically. Viewed another way, a nondeterministic machine running in time t on input x starts running 2^t computations simultaneously in parallel (one computation for each different choice of the "magic word" m), and accepts if *any* of the 2^t computations outputs 1. It is useful to think of the "magic word" m as a "proof" that the input x should be accepted.

On the face of things, this looks like a really goofy model of computation. But it is exactly the right model of computation to use, if we want to understand a host of important computational problems.

Definition 4 *The class* NP *consists of all computational problems A for which there is a polynomial p and a nondeterministic machine running in time $p(n)$ on inputs of length n, that accepts input x if and only if $x \in A$.*

A few hundred of the most important NP-complete problems can be found in the standard reference work by Garey and Johnson [38]. Some of the most familiar of these are:

- SAT (the set of Boolean formulae that have a satisfying assignment).
- CLIQUE (the set of pairs (G, k), where G is a graph for which there is a subset of k vertices, all of which are connected to each other).
- 3-COLORABILITY (the set of graphs G whose vertices can be colored Red, Green, and Blue, such that no edge has endpoints with the same color).

Note that, for these three examples, it is easy to see what the "magic word" would be that provides a proof of membership. For SAT it would be a satisfying assignment (which can easily be checked to see that it is, indeed, a satisfying assignment); for CLIQUE it would be a set of k vertices; for 3-COLORABILITY it would be an assignment of colors to the vertices.

It is hard to overstate the usefulness of NP-completeness as a tool for understanding the apparent intractability of many problems that we would dearly love to be able to solve with computers. In 1997, Papadimitriou wrote [78]:

> *... about 6,000 papers each year have the term "*NP-*complete" on their title, abstract, or list of keywords. This is more than each of the terms "compiler," "database," "expert," "neural network," and "operating system." Even more surprising is the diversity of the disciplines with papers referring to "*NP-*completeness:" They range from statistics and artificial life to automatic control and nuclear engineering.*

Many other important computational problems do not seem to be NP-complete, but are complete for complexity classes defined using variations on the theme of nondeterminism (such as *counting* the number of different proofs of acceptance, instead of merely asking if such a proof exists [91]).

One can easily establish the following inclusions:

$$\text{P} \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{EXP}.$$

Thus, just as we obtain no *proof* of intractability from the knowledge that a problem is PSPACE-complete, so also a proof of NP-completeness yields no proof that a problem is hard to compute, but the evidence of intractability is nearly as compelling for NP-complete problems as it is for PSPACE-complete problems. In practice, this turns out to be very useful information.

It is known that P is not equal to EXP, and thus at least one of the three inclusions above must be a proper containment, but it is not known that any one of them is proper. Most people working in the field would probably conjecture that *all* of these containments are proper – but it is risky to rely on this sort of "intuition". Later in this paper, we discuss one important example, where two classes that were thought to be distinct turned out to be the same.

One strange thing about the nondeterministic model of computation is that the tasks of accepting and rejecting an input are very different. Given any problem A, the complementary problem $\overline{A} = \{x : x \notin A\}$ is in P if and only if $A \in$ P. This does not seem to be true for NP. Consider $\overline{\text{SAT}}$; nobody has any idea how to give a short proof that a formula x does *not* have a satisfying assignment. This gives rise to the complexity class coNP $= \{A : \overline{A} \in \text{NP}\}$. Clearly P $\subseteq$ NP $\cap$ coNP.

2.6 Subclasses of P

Polynomial-time reducibility is not very useful for drawing distinctions between problems in P – but useful distinctions can be made. What is needed is a more delicate tool, defined analogously to polynomial-time reducibility, but using a more restricted class of functions. We will not provide definitions here, but merely note that logspace reducibility [57] is frequently used to define classes of complete problems inside P, as are even more restrictive notions of reducibility, defined in terms of small circuit classes [7, 4]. More information on subclasses of P can be found by consulting the references [44, 93, 8].

With very few exceptions, natural problems that are complete for NP, PSPACE, and EXP (and other complexity classes) under polynomial-time reducibility remain complete even when these more restrictive notions of reducibility are used instead. Thus, there is essentially no drawback to using the more restrictive notions of reducibility (since the problems that one wants to classify remain complete under the more restrictive reducibility, and as a bonus one is able to show that certain problems are complete for P and others are complete for interesting subclasses of P).

When one is first introduced to the notions of NP-completeness and completeness in other complexity classes, it probably seems as if completeness is a very unusual property, and that it should be rare for a problem to be complete for a complexity class. However, the opposite is true. The surprising lesson that emerges, after decades of experience in complexity theory, is that *the overwhelming majority of natural computational problems can be shown to be complete for one of about a dozen or so complexity classes.* Sometimes the definitions of these complexity classes may seem cumbersome or complicated (just as the definition of NP may strike one as being fairly unnatural at first). However, it is important to note that these classes are "*discovered*, and not *invented*" (quoting Papadimitriou again [78]). That is, it is *not* the case that some out-of-touch theoretician cooked up the definition of these complexity classes merely to prove a few theorems. Rather, there was an important class of computational problems out there that people were already interested in, and then complexity theoreticians were able to show that the problems were all in fact complete for some complexity class that could be described in terms of resource bounds applied to some computational model.

Let us return again to the question that begins Section 2:

2.7 What is the ‘P=NP?’ Problem?

We have introduced P and NP, so perhaps the answer is clear: The ‘P=NP?’ problem is simply the question of whether those two complexity classes are one and the same. But P and NP are just two of the most important complexity classes. The P vs NP question really stands for a more fundamental and general question concerning the nature of this entire framework of complexity classes, with its partition of natural problems into classes of complete sets for various complexity classes. How much of this structure is real, and how much is an illusion? The theory of complexity classes seems to explain our inability to find fast programs for certain problems – but is this explanation real, or is it simply a convenient and comforting tale that we tell ourselves? We find it comforting, because it would mean that the reason we have not found a fast algorithm is *not* because we are too stupid to find it – it is because no fast algorithm is possible.

3 Why is the ‘P=NP?’ Problem Important?

Once again, it seems that we are asking a question that we have already answered. The authors of the 6,000 papers per year that mention NP-completeness have their own reasons for wanting to know if these complexity classes are equal or not. The people who are trying to solve NP-complete optimization problems have a powerful financial incentive for wanting to know if these problems have fast algorithms or not. These are all significant reasons for why the P vs NP problem is important – but there are some additional reasons that should be discussed, too. That is the task we take up in the rest of this section.

3.1 Cryptography

Modern cryptography relies on the existence of *one-way functions* (functions f that are easy to compute but have the property that no efficient algorithm can take as input a string y in the range of f and produce as output a string x such that $f(x) = y$). Actually the requirements are much stronger; it is necessary that the task of finding such an x be difficult *on average*, so that for the *overwhelming majority* of the strings y in the range of f, there is no way to find x such that $f(x) = y$. For instance, the RSA cryptosystem [80] relies on multiplication being a one-way function; define f to be the function that takes as input two prime numbers u and v each having the same number of bits, and produces uv as output. If there is an efficient algorithm that can find the prime factors of a given integer, then f is not one-way and many cryptographic applications in wide use are insecure.

If P $=$ NP, then there would seem to be no way to salvage cryptography. The problem of inverting a supposed one-way function lies in NP; if all of these problems are in P, then in order to rescue the notion of a one-way function one would have to hypothesize the existence of functions that are very easy to compute, but whose inverses require time n^k for very *large* values of k. Complexity theory currently offers no suggestions as to how this might be accomplished.

But in fact, complexity theory currently offers few if any useful tools that can be used to provide evidence that a given function is a cryptographically secure one-way function. Let me elaborate on this point. If we know that a problem is NP-complete, then there is a coherent theoretical framework explaining why the problem is probably hard to compute; if the problem turns out to be easy, then the entire framework comes crashing down. In contrast, consider the factoring problem. The best evidence that factoring is hard comes from the fact that people have been trying to find good factoring algorithms for a few hundred years, without success. This is not particularly strong evidence, since there are several notable examples in which problems not previously known to reside in P have yielded to new algorithmic insights and techniques. If factoring turns out to be easy, it will have dramatic consequences for the practice of cryptography, but it will not fundamentally alter the framework of complexity theory. Similar observations hold for all of the widely-considered candidate cryptographically-secure one-way functions.

Another factor to keep in mind is that, if f is a *one-one* function, then the problem of computing the inverse of f lies in NP $\cap$ coNP [22]. Thus in order to have one-one one-way functions, one needs not only P $\neq$ NP, but P $\neq$ NP $\cap$ coNP which seems to be a stronger hypothesis.

An alternate approach might be to design a function f that is "complete" in some sense for the class of all one-way functions (so that f is cryptographically secure if any function is). In fact, such a construction was presented already by Levin [66], although this construction holds little interest for practitioners, since the constants in the security guarantees are quite weak.

Recall that this is the section of the paper in which we are addressing the question of why the P vs NP question is important. In order for modern cryptography to rest on a firm foundation, a proof that P $\neq$ NP is an absolutely essential first step – but it would be only a first step. Much stronger intractability results are required for cryptography.

3.2 Understanding the World

There are few areas of scientific enquiry that are untouched by algorithmic considerations. In biology, economics, and physics, many of the natural processes that are studied can be viewed as proceeding algorithmically. If a biological theory (or an economic theory, a theory of evolution, etc.) requires that an organism (or participants in a market, or an environmental system, respectively) arrive at an optimal state, then it had better be the case that this does not require the solution of an intractable computational problem, or else the theory lacks plausibility. But until we know if P $=$ NP, we don't have a good understanding of which problems are intractable. Aaronson has proposed hypothesizing the intractability of NP-complete problems as a natural law, in order to judge the plausibility of certain interpretations of quantum mechanics [2]. If certain aspects of a physical theory can be shown to imply that there are efficient ways to solve NP-complete problems, then this should throw doubt on the theory (since there is no empirical evidence that these problems are amenable to efficient solution). But how compelling can such an argument be, until we know for sure that conventional programs and computers are unable to solve NP-complete problems?

The P vs NP problem can be cast as the problem of whether it is significantly harder to find a proof of a theorem, than to merely check that a proof is correct, and thus it has profound implications for mathematics. That is, if we consider any fixed formal proof system (so that the problem of checking if a proof is correct is simply a syntactic procedure that can be automated), then the set of theorems that have proofs of a reasonable length is a problem in NP. Stated another way, if you want to know if there is a proof of some statement ϕ that is at most 60 pages long, a nondeterministic machine can determine the answer quickly (where the "magic word" is simply the 60 page proof). If P $=$ NP then there is a relatively fast automatic way to find the 60 page proof, given only statement of the theorem. Long before the P vs NP question was formalized, Gödel and von Neumann discussed precisely this scenario [45]. The connection between complexity and the notion of proof has a long history, and has played a crucial role in some of the most dramatic developments of the theory, as discussed later in this paper.

For all of these reasons and more, there can be little question that the P vs NP question is important.

4 What Progress Has Been Made in the Past Thirty Years?

This is the most dangerous section of the paper to write. The dangers are (at least) twofold:

- I can omit some really important developments.
- I can get carried away about one or the other line of research and include more detail than I should, resulting in a long, unreadable document.

In order to steer a course between Scylla and Charybdis, I will try to keep the paper short and easy to read, even though this means that I will leave out some great stories (and my treatment of the stories that I include will be far too brief).

4.1 Small Circuits and the Polynomial Hierarchy

In order to present some of the exciting developments of the past three decades, I need to present a bit of material that is slightly older.

In a paper published in 1976 [88], Stockmeyer introduced a hierarchy of complexity classes that sits "right above" NP. We have already been introduced to the lowest levels in this hierarchy: P, NP and coNP. A number of problems related to NP optimization problems are more conveniently stated in a form that is hard for both NP and for coNP and thus is not believed to lie in either class. For example, consider the Traveling Salesperson Problem. This can be phrased as a problem in NP in the following form: Given a graph G with "distances" on the edges and a number k, is there a path of length at most k that visits all of the vertices in the graph? But one might find it more natural to ask "Given G, compute the length of the shortest path that visits all of the vertices." If one had a subroutine for the NP formulation of the Traveling Salesman Problem, then this value could easily be computed using binary search. Thus it lies in the class P^{NP} (the class of problems that can be solved in polynomial time using an "oracle" – that is, a subroutine whose running time we do not count – for a problem in NP). In fact, this turns out to be a complete problem for P^{NP} [94].

Once we have defined P^{NP}, it is a short step to define $\mathrm{NP}^{\mathrm{NP}}$ and $\mathrm{coNP}^{\mathrm{NP}}$. These classes seem to bear the same relationship to P^{NP} as NP and coNP bear to P, and thus it seems that these classes provide substantial additional computational power over P^{NP}. There are natural and well-studied problems that are complete for $\mathrm{NP}^{\mathrm{NP}}$ and $\mathrm{coNP}^{\mathrm{NP}}$. This process can be continued, to define an infinite hierarchy of complexity classes, known as the *polynomial hierarchy*. One property of this hierarchy is that, if any two levels coincide, then the entire hierarchy collapses to that level. Thus if $\mathrm{NP}^{\mathrm{NP}} = \mathrm{coNP}^{\mathrm{NP}}$, then the entire hierarchy collapses to $\mathrm{NP}^{\mathrm{NP}}$; if $\mathrm{P} = \mathrm{NP}$, then the hierarchy collapses to P. One reason the polynomial hierarchy has come to be important in the field of complexity theory, is because the belief that the hierarchy is infinite is nearly as well-rooted as the belief that $\mathrm{P} \neq \mathrm{NP}$.

The story of small circuits for NP illustrates this use of the polynomial hierarchy.

Recall from Section 2 that there are two basic models of computation: programs and circuits. With programs, there is one program that works for all input lengths; with circuits, there is a different circuit for each input length. Of course, if you have an efficient way to *build* circuits for your problem, there is not much difference between these two notions. But there are many problems that have *no program at all*, but which have small circuits (e.g., consider unary encodings of the Halting Problem).

Circuit complexity is essential, in order to prove that certain transformations from input to output *on a fixed input size* are intractable. This is an important point, and it is worth emphasizing.

Consider for example the following theorem regarding the problem of determining whether a logic formula in a certain formalism (abbreviated WS1S) is true or not.

Theorem 5 *[89] Any circuit of* AND, OR, *and* NOT *gates that takes as input a WS1S formula of 610 symbols and outputs a bit that says whether the formula is true must have at least* 10^{125} *gates.*

Clearly, no such circuit can fit in the solar system. Note that this theorem is quite specific about saying that this problem is difficult at a particular input length. This is the sort of security guarantee that is required in cryptography, where one needs to pick key lengths long enough so that the problem of cracking the system is intractable; it is not enough to know that programs will require a long amount of time for "big enough" inputs – it is important to know just how big is "big enough".

Recall the Checkers problem (which is complete for EXP) from Section 2.4. We know that any program that solves Checkers must run for time 2^{n^ϵ} for *large* inputs – but we have absolutely no proof that this problem does not have circuits of *linear* size! (If this is the case, then there is *no* input size where Checkers becomes intractable.) For all we know, perhaps all problems in EXP have circuits of polynomial size.

Thus for example, if cryptographers are ever to have any hope of using theorems of complexity theory to pick key sizes to make their cryptosystems secure, it is essential to know not only that P $\neq$ NP, but that NP does not have polynomial-size circuits.

But is this question really any different from the P vs NP question? There is a long line of research that tries to relate these problems, beginning with a result by Karp and Lipton [59] showing that NP has polynomial-size circuits only if the polynomial hierarchy collapses to $\mathrm{NP}^{\mathrm{NP}}$. There has been work through the years trying to show that the collapse happens at a lower level [24, 25], but it is still not known if NP having polynomial size circuits implies that the polynomial hierarchy collapses to P^{NP}.

There are many results in complexity theory of the form "X is true, unless the polynomial hierarchy collapses". This is taken as strong evidence that X is true.

4.2 Interactive Proofs and Probabilistically Checkable Proofs

There is widespread agreement that the most significant change in our understanding of NP over the last three decades grew out of an expanded notion of "proof" [16, 42]. Recall that NP can be viewed as the class of sets B for which there are short "proofs" that $x \in B$, where a "proof" is just a string y so that a polynomial-time program A can read the pair (x, y) and accept if and only if y provides the information that is necessary to convince the program that x is in B. Recall from Section 2.2 that there is a segment of the community that contends that *probabilistic* polynomial time algorithms are a more appropriate way to capture the notion of "feasible computation." Starting from this point of view, it seems natural to consider an expanded version of NP, defined in an analogous way, but allowing a *probabilistic* algorithm A to determine if y provides convincing evidence that x is in B. This leads to a complexity class called MA; the name comes from viewing the process of proving membership in B as being a conversation between a magical "prover" (Merlin the Wizard, who provides the string y) and a mere mortal with limited computational power, but owning a pair of dice to help him make random choices (King Arthur). Viewing things this way, Merlin speaks first, giving Arthur the string y, and then Arthur rolls his dice and does his computation on the pair (x, y).

But is there any reason why Merlin should have to speak first? One can also define the class AM, where Arthur gets string x, rolls his dice to get some random bits, and on the basis of these bits poses a question to Merlin, who then sends a string y (and Arthur can do some computation to see if Merlin's reply convinces him). One can show that MA $\subseteq$ AM. There seems to be no reason to stop at such short conversations; one can define an entire sequence of classes AMA, AMAM, . . ., as well as a class IP (for "Interactive Proofs") where the conversation continues for a polynomial number of rounds.

Why is this interesting?

One interesting fact is that AM $=$ AMA $=$ AMAM, etc. That is, two rounds of communication are as good as any constant number of rounds [16].

Much more interesting is the fact that there is an important computational problem in AM that is not known to lie in NP: the graph non-isomorphism problem [42]. The graph isomorphism problem (given two graphs G and H, are they isomorphic?) is easily seen to be in NP. (The "magic word" is simply a permutation of the vertices of G, yielding the graph H.) But how can you provide a short proof that two graphs are *not* isomorphic? If Merlin is all-powerful, then here's how Arthur can be convinced. Arthur gets the graphs G and H, and picks one at random, permutes the vertices in a random way and obtains a new graph K_1. He repeats this process and obtains a graph K_2, etc., until he has 100 graphs $K_1, \ldots K_{100}$. Arthur knows, for each K_i, if K_i is a copy of of G or of H. Arthur now sends the sequence $K_1, \ldots, K_{100}$ to Merlin, and asks Merlin to tell him which graph each K_i is a copy of. Note that if G and H are not isomorphic, the all-powerful Merlin has no problem doing this. But if G and H are isomorphic, then Merlin has just a 1-in-2^{100} chance of sending Arthur the right answer. Thus if Merlin sends Arthur the correct answer, Arthur is justified in feeling quite confident that G and H are really not isomorphic.

Much of the initial interest in this type of "proof" came from the notion of "zero-knowledge proofs" which has wide application in cryptography. (This comes from the observation that Arthur gains no useful information in the preceding example, other than being convinced that the graphs are not isomorphic.) This is a huge topic that is surveyed elsewhere (e.g., [98, 41]).

The class IP was something of a mystery for a while. It was felt that this notion of "proof" was probably not too much stronger than the usual notion of "proof", and many felt that it would be unlikely that IP would contain coNP. In fact, evidence was presented that was considered at the time to be rather compelling, arguing that "new techniques" would be required to show coNP $\subseteq$ IP [36].

New techniques *were* found, and a dramatic series of papers ended by showing that IP $=$ PSPACE [68, 83]!

According to the rules of IP interaction, Merlin is allowed to give different responses to the same query from Arthur. That is, if Arthur has random sequence r_1 that causes him to ask query q at some point during his interaction with Merlin, Merlin might give a different response than he gives during the interaction that Arthur has with him when using random sequence r_2. If the rules of the game are changed, so that Merlin has to commit ahead of time to the response that he'll give to each possible query q from Arthur, this turns out to give a characterization of nondeterministic exponential time (NEXP) [14]. This is an *extremely* counterintuitive characterization. Nondeterministic exponential time can be viewed as the class of problems B where membership of x in B can be demonstrated by a "proof" of length exponential in the length of x. This characterization means that Merlin can provide an exponential-sized proof, and Arthur can be convinced of its correctness by randomly picking just a small number of positions in the proof to examine!

Soon a similar characterization of NP was given, and various parameters in the characterization were optimized, to obtain a truly spectacular and almost unbelievable re-working of the notion of "proof"

[12, 13]. Rather than give a formal definition of a "Probabilistically Checkable Proof," let us give an example. Suppose that you are asked to referee a paper, but you're very short of time. Rather than read the entire paper, you randomly pick a few paragraphs, and if you don't see any problem, then you decide to accept the paper. Most of us would consider this to be very irresponsible behavior. How can one have any confidence in the correctness of a proof without reading every symbol of the proof? But in fact *any* proof can be encoded in such a way that this sort of "lazy" refereeing is sufficient. That is, the proof can be encoded as a string y (of length not much greater than the length of the original proof), so that a verifier can randomly choose k bits of y (where k is a constant that does *not* depend on the length of y, and the verifier is using only $O(\log n)$ random bits in order to make this selection), and will always accept if the proof is correct, and will reject with very high probability if the proof is not correct [12, 13]. The proof of this characterization is one of the most complicated arguments in complexity theory, and there has been a great deal of interest in finding a simpler proof; substantial simplifications have been presented in the last few years [31, 50].

4.3 Hardness of Approximation

Since the CLIQUE problem is NP-complete, we know not to expect to be able to find the size of the largest clique in a graph. When we learn that it is hard to find an optimal solution, it is natural to adjust our goals, and settle for getting the best solution that we can. There is a huge literature on algorithms for finding approximate solutions to NP-complete optimization problems, and there were some early results showing that certain approximations could not be obtained efficiently unless P $=$ NP (for example [39]). For many years, however, there was little known about which approximations could be obtained efficiently, and which ones are intractable.

The dramatic characterization of NP in terms of probabilistically-checkable proofs changed all of that. Given a probabilistically-checkable proof, it is either the case that *all* of the polynomially-many probabilistic sequences lead to acceptance, or only a *small fraction* of them do. It turns out to be possible to build on this, to reduce CLIQUE to instances where there is either a very large clique, or else the largest clique is quite small, and thus CLIQUE is hard to approximate unless P $=$ NP [33, 47]. Related approaches work on many other optimization problems in NP, and in some cases it is possible to give tight bounds, showing that a solution can be found that is at most α times the optimal for some constant α, but doing any better is NP-hard. (See [48] for one such example. There are many others.)

4.4 AM **and** NP

We discussed the Arthur/Merlin class AM in Section 4.2. It is safe to say that, back when AM was introduced [16, 42], most people in the field were inclined to think that AM was likely to contain problems that were outside of NP. In the intervening years, there was astonishing progress made in the field of *derandomization* (that is, the study of eliminating the use of probabilistic bits in randomized algorithms). Since this survey focuses on NP, I will limit the discussion here to the probabilistic analog of NP, which is AM. Let us learn what has happened in the last two decades, to change perceptions of the likely relationship between AM and NP.

A defining characteristic of a random coin toss is that it is *unpredictable*. The outcome of a hard-to-compute function is also "unpredictable" in some sense (although it is not clear that there is a meaningful connection between these two settings, since a function always gives the same answer to a given question, unlike tossing a coin). A sequence of important papers (see [75, 55]) showed that this connection can be made precise and exploited. If there is a problem A that is computable in time 2^n that requires *circuits* of size $2^{\epsilon n}$ for some $\epsilon > 0$ (that is, if A requires circuits of nearly maximal size), then computing A on all of the inputs of size $O(\log n)$ can be used to produce a sequence of bits that is enough like "noise" that it can be used to give a deterministic simulation of a probabilistic algorithm.

Related techniques were also applied [60, 73, 82] to generate random bits that could be used to give nondeterministic simulations of AM. These techniques led to the conclusion that NP $=$ AM if there is a problem computable in nondeterministic time 2^n that requires *nondeterministic* circuits of size $2^{\epsilon n}$ for some $\epsilon > 0$. We will not define nondeterministic circuits here – but we will mention that this hypothesis is considered to be reasonably likely, and hence much of the complexity-theoretic community would now conjecture that AM $=$ NP. Note that this also would imply that the graph isomorphism problem is in NP $\cap$ coNP.

4.5 Average-Case Complexity

Even if we assume that P $\neq$ NP and hence we must give up on the idea of having efficient algorithms that solve NP-complete problems, there is still a pressing need to have algorithms that perform as well as possible in solving these problems. Various heuristics have been developed for different NP-complete problems that seem to perform reasonably in various settings, and one occasionally hears the claim that a particular heuristic "works well for instances that arise in practice".

Such claims can be difficult to evaluate, since it is usually very difficult to say anything precise about the probability distribution on inputs in real-world settings. Sometimes it is useful to talk about the performance of an algorithm when inputs of a given length are distributed uniformly, but it is easy to give examples of problems that are very easy to solve using the uniform distribution. For example, consider the set $\{(\phi, \phi) : \phi \in \text{SAT}\}$. From one perspective, this is just a simple encoding of SAT – but from another perspective, this set is trivial to solve on all but an exponentially-small fraction of the inputs (since we need only reject if the first half of the string is different from the second half, and this will almost always be the case).

Levin introduced a theory of average-case complexity [65], and presented an NP-complete problem that is hard on average to solve, using *any* "reasonable" distribution on the inputs. (It is necessary to restrict attention to some class of distributions, since – assuming that P $\neq$ NP – any efficient algorithm will make errors on *some* inputs, and there is always an "unreasonable" probability distribution that places all of its weight on those inputs where the algorithm fails.) Levin focused on distributions that are computable in polynomial time. Analogous studies have been carried out, based on so-called "samplable" distributions, which are distributions that can be "generated" by a probabilistic polynomial-time algorithm. This model makes sense, if you hypothesize that the input instances that arise in practice are in fact generated by some feasible process. There are some excellent surveys of this type of approach to average-case complexity [95, 96, 53].

Although the uniform distribution is not always the most relevant distribution to consider, some very significant insights have been gained by considering how well algorithms can perform on the uniform distribution. A crucial step in some of the derandomization arguments that were considered in Section 4.4 was the proof that, if there is any problem in EXP that is not solvable by polynomial-size circuits, then there is a problem in EXP for which any polynomial-size circuit gives the wrong answer for nearly half of the inputs of length n [15]. This is called a "worst-case-to-average-case" reduction, because it involves showing how to compute a function A correctly on *all* inputs, by accessing any circuit that computes a related function A' correctly on a large fraction of the inputs. This sort of argument draws heavily on the theory of error-correcting codes; the truth table of A' is essentially an encoding of the truth table of A using an error-correcting code. A circuit that computes A' correctly on a reasonably large fraction of the inputs can be viewed as a corrupted version of the codeword – but there are a small enough number of errors so that the truth table of A can be recovered.

Related worst-case-to-average-case reductions are known for NP [76, 49, 90, 43], although the parameters are not as good as in the corresponding results for EXP, because of the additional technical obstacles that arise when working with NP.

4.6 Time-Space Tradeoffs

Proving that P $\neq$ NP involves proving a superpolynomial lower bound on the run time of any algorithm for SAT. Is there any way to measure our progress toward this goal? For instance, do we know that SAT requires time n^3, or time $n \log n$?

Sadly, the answer is "No." We still do not know if SAT can be recognized in *linear* time on a Turing machine. However, a series of papers beginning with [35] (and nicely surveyed by Van Melkebeek [92]) shows that algorithms for SAT that use *small space* must run for time more than $n^{1.8}$ [99]. (These results hold not only for Turing machines, but for more general models of computation that allow random access to memory locations.)

4.7 The Isomorphism Conjecture

All NP-complete problems are equivalent in some sense. Berman and Hartmanis noticed that all of the NP-complete problems in the monograph by Garey and Johnson [38] in fact are *isomorphic* to each other, in a very strong sense [18]. Namely, they showed that, for any two of these problems A and B, there is a *bijection* f such that both f and f^{-1} are computable in polynomial time, mapping A onto B. Thus, in a natural and appealing way, it is reasonable to say that all of the NP-complete problems in Garey and Johnson are simple re-encodings of each other. They conjectured that, in fact, this is true for *all* NP-complete problems, and not merely the ones in Garey and Johnson.

If true, this would of course imply P $\neq$ NP, since if P $=$ NP there are *finite* sets that are NP-complete.

The Berman-Hartmanis conjecture fueled interest in the general question of just what what can be proved about what NP-complete sets must "look like". For instance, they cannot be finite unless P $=$ NP, but can they have a "small" number of strings? If the isomorphism conjecture is true, any NP-complete set must have at least 2^{n^ϵ} strings of infinitely-many lengths n, for some $\epsilon > 0$. Can one prove that all NP-complete sets must have this many strings (assuming P $\neq$ NP)? Can there be *sparse* NP-complete sets (i.e., sets with at most a polynomial number of strings of each length)? We now have a fairly clear answer to these questions.

Mahaney's Theorem [69] says that there are sparse NP-complete sets if and only if P $=$ NP. Some years later, Ogihara and Watanabe gave a simpler proof of this theorem that also extends to a larger class of reducibilities [77]. Very recently, Buhrman and Hitchcock proved that the 2^{n^ϵ} bound (i.e., the bound that is implied by the isomorphism conjecture) is tight, unless the polynomial hierarchy collapses [23].

In spite of theorems such as this that seem to support the isomorphism conjecture, there seems to be little confidence these days that the conjecture is true. For example, if f is a cryptographically-secure one-way function, the set $f(\text{SAT})$ does not appear to be isomorphic to SAT. There are a number of excellent surveys of work on the isomorphism conjecture, including [61, 70, 100].

Interestingly, when more restrictive notions of reducibility are considered, the isomorphism conjecture can be replaced by an isomorphism *theorem*. Recall from Section 2.6 that, in investigating subclasses of P it is useful to consider reductions computed by restricted classes of circuits. (These are known as AC^0 reductions.) With very few exceptions, natural problems that are known to be complete for some complexity class under any kind of reducibility can be shown to be complete under AC^0 reductions; thus the framework of complete sets that we use in order to understand the complexity of natural problems can be formulated entirely in terms of AC^0 reductions.

It turns out that for all complexity classes of interest, *all* the sets that are complete under AC^0 reductions are isomorphic under bijections f such that both f and f^{-1} are computable in AC^0 (and in fact the bijections can be shown to have a very restricted form) [7, 6, 4]. Thus some form of the isomorphism conjecture turns out to be true.

5 Where Are We Now? (Barriers to Progress)

For many years, it was felt that radically new techniques would be needed, in order to make any significant progress on the P vs NP problem. This was because the "traditional" techniques in the complexity theorist's toolkit all "relativized". What does this mean?

Recall the notion of having "free" access to a problem as a subroutine or "oracle," as discussed in Section 4.1. If you have a class of programs or machines that characterize a complexity class such as P, NP, or EXP, and you provide each such machine with an oracle for problem A, one obtains the new "complexity classes" P^A, NP^A, and EXP^A. Baker, Gill, and Solovay [17] observed that all of the theorems that were proved using the usual proof techniques of the period (for example, the theorem: $\text{P} \neq \text{EXP}$) would carry over relative to every oracle (and hence, for every A, $\text{P}^A \neq \text{EXP}^A$). They also showed that there were sets A and B such that

- $\text{P}^A \neq \text{NP}^A$).
- $\text{P}^B = \text{NP}^B$).

In fact, the set B can be chosen to be any PSPACE-complete set.

Over the next several years, a great many open problems in complexity theory were shown to admit contradictory relativizations in this sense. The general sense in the community was that it was generally hopeless to spend time on such questions, since they obviously required non-relativizing proof techniques, and nobody had a good idea about how to develop such techniques. For instance, when Fortnow and Sipser presented an oracle A relative to which $\text{coNP}^A \not\subseteq \text{IP}^A$ [36], it convinced several people to stop thinking about trying to show that IP contained the polynomial hierarchy.

5.1 Nonrelativizing Proof Techniques

Thus the community sat up and took notice when it was shown that $\text{IP} = \text{PSPACE}$ [68, 83]. Finally, there was a fundamentally new set of tools to apply!

There followed an intense period of activity, where several new non-relativizing theorems were proved. (Since the focus of this paper is on NP and I want to avoid introducing new complexity classes, I will avoid describing these in more detail.) Recently, Aaronson and Wigderson took up the challenge of characterizing these "new" proof techniques, and determining what their limits are [3]. They define a new notion called "algebrization" (which I will not define here) and show that essentially all of the results that have been proved using non-relativizing proof techniques "algebrize". They also show that the P vs NP problem cannot be resolved by any algebrizing proof technique (nor can the problem of showing that nondeterministic *exponential* time does not have circuits of polynomial size).

So once again we are in the position of needing fundamentally new proof techniques in order to make progress on some of the big open questions, but at least we once again know what some of the barriers are (and we also have received a healthy jolt of optimism from the experience of seeing the barrier of relativization fall a number of years ago). We shall overcome!

5.2 Natural Proofs

I would be dishonest if I were to give the impression that there is great optimism that we are on the verge of a breakthrough that will finally resolve the big open problems in complexity theory. There are many barriers to progress that have been identified.

Razborov and Rudich studied the approaches that have been followed in proving superpolynomial circuit size lower bounds on restricted classes of circuits, and observed that these approaches all fall into a certain "natural" approach to trying to prove circuit lower bounds [79]. They also proved that, if cryptographically secure one-way functions exist, then no such "natural proof" can prove that problems in NP require circuits of more than polynomial size. At one level, this was demoralizing, since it explained why some fairly modest-sounding lower bounds cannot be obtained without formulating a fundamentally new approach. On the other hand, work such as this can serve as a useful road map, helping the community to plan its assault on the big open questions in complexity theory.

There have been a number of suggestions of possible strategies to avoid the pitfalls represented by Natural Proofs and Algebrization [5, 74, 10, 37, 26]. I discuss some of these at more length (and provide more background and motivation) in a recent survey [9].

6 Conclusions: What Would a Solution Mean?

What would it mean to "solve" the P vs NP problem? How can one claim the $1 Million Dollar prize offered by the Clay Mathematics Institute [27]?

There seem to be three possible solutions (listed in my personal order of preference):

1. Prove that P $\neq$ NP.
2. Prove that P $=$ NP.
3. Prove that there is no proof one way or the other.

Let us deal with the last option first. It is certainly the most frustrating possibility of the three. Imagine if a fast program for SAT *exists*, but there is no way to prove that it actually works correctly! (You could use such a program to provide satisfying assignments whenever it claimed that a formula was satisfiable – but how could you know it was correct in claiming that a formula was *not* satisfiable? Of course, in this case it would follow that NP $=$ coNP and thus there is some sense that there are short "proofs" of unsatisfiability — but there might be no way to "prove" that these "proofs" were doing what they claimed.) I encourage the reader who wants to learn more about this possibility to read the entertaining survey on this topic written by Aaronson [1].

One point that Aaronson makes is that it is unlikely that a proof of this third possibility will surface any time soon. This is because current approaches to proving that a statement ϕ is independent of some formal system $\mathcal{S}$ almost always prove that ϕ is actually independent of the stronger system that results by augmenting $\mathcal{S}$ by all *true* first-order logic statements that contain only universal quantifiers. This is relevant, because it is known that if "P $\neq$ NP" is independent of this stronger formal system, then one can show that SAT must have circuits of "almost" polynomial size. (Namely, it has circuits of size $n^{\alpha(n)}$ where α is a *very* slow-growing function [62].) That is, if "P $\neq$ NP" is independent of this stronger formal system, then it's almost the same as SAT being easy to compute anyway.

Let us move on to the second possibility: P $=$ NP. At one level this would be very disheartening because it would mean that the entire framework of completeness that seemed to explain so much was nothing but a glorious illusion. The optimistic way for a proof of P $=$ NP to occur would actually yield an *efficient* (say, linear or quadratic time) algorithm for SAT. In this case, the consequences would be stunning. Mathematics could be automated. Machine learning and other tasks in artificial intelligence would become trivial. Corporate efficiency would soar as all sorts of optimization problems suddenly would become routine. Cryptography would become impossible as currently conceived. Impagliazzo describes this possible world as "Algorithmica" [53], and the reader is encouraged to read his description.

. . . but this assumes that SAT has a truly efficient algorithm. The more pessimistic possibility that might emerge from a proof of P $=$ NP is an algorithm for SAT with a running time of n^{1000}. Worse yet would be a nonconstructive proof, showing that a polynomial-time algorithm for SAT *exists* but providing no clue as to how to find such an algorithm. (There is precedent for nonconstructive proofs that problems can be solved in polynomial time [34], so this possibility cannot be dismissed out of hand.) Historically, when natural problems have been shown to lie in P, in almost all cases reasonably efficient algorithms have eventually been found. Problems that really require time n^{1000} to solve seem to be exotic oddities that nobody would really want to try to compute anyway; natural problems seem to either have efficient algorithms or require essentially exponential time. This optimistic belief is really predicated on the computational universe not being truly perverse. We have no proof that this optimism is warranted.

Finally, let us consider the preferred outcome: someone finds a proof that P $\neq$ NP. Again, there are several possibilities to consider.

One possibility is that P $\neq$ NP but that every NP-complete problem is easy on average in the sense of [65]. This means that, for every fast algorithm for SAT there are some instances where the algorithm gives a wrong answer – but these instances essentially never come up in practice so you don't really notice it. This corresponds to the possible world that Impagliazzo calls "Heuristica" [53]. It might seem as if this outcome is indistinguishable from the case where P $=$ NP, but as Impagliazzo points out [53], if P $=$ NP then every problem in the polynomial hierarchy has a polynomial-time algorithm. In contrast, if SAT is easy on average, it is not clear that the same is true for problems in the polynomial hierarchy.

Even if we're really lucky and a proof of P $\neq$ NP shows that SAT requires nearly exponential time, note that much, much more is required for some of the important applications that rely on intractability lower bounds (such as cryptography). At the very minimum, we would need *circuit size* lower bounds, in order to talk about intractability for any given input size (as discussed in Section 2.4).

A proof that P $\neq$ NP would not be the end of the story. It would only be the beginning.

References

[1] S. Aaronson. Is P versus NP formally independent? *Bulletin of the EATCS*, 81:109–136, 2003.

[2] S. Aaronson. Guest column: NP-complete problems and physical reality. *SIGACT News*, 36(1):30–52, 2005.

[3] S. Aaronson and A. Wigderson. Algebrization: A new barrier in complexity theory. In *Proc. ACM Symposium on Theory of Computing (STOC)*, pages 731–740, 2008.

[4] M. Agrawal. The first-order isomorphism theorem. In *Proc. Conference on Foundations of Software Technology and Theoretical Computer Science (FST&TCS)*, volume 2245 of *Lecture Notes in Computer Science*, pages 70–82, 2001.

[5] M. Agrawal. Proving lower bounds via pseudo-random generators. In *Proc. Conference on Foundations of Software Technology and Theoretical Computer Science (FST&TCS)*, volume 3821 of *Lecture Notes in Computer Science*, pages 92–105, 2005.

[6] M. Agrawal, E. Allender, R. Impagliazzo, R. Pitassi, and S. Rudich. Reducing the complexity of reductions. *Computational Complexity*, 10:117–138, 2001.

[7] M. Agrawal, E. Allender, and S. Rudich. Reductions in circuit complexity: An isomorphism theorem and a gap theorem. *J. Comput. Syst. Sci.*, 57:127–143, 1998.

[8] E. Allender. Arithmetic circuits and counting complexity classes. In J. Krajíček, editor, *Complexity of Computations and Proofs*, volume 13 of *Quaderni di Matematica*, pages 33–72. Seconda Università di Napoli, 2004.

[9] E. Allender. Cracks in the defenses: Scouting out approaaches on circuit lower bounds. In *Computer Science – Theory and Applications (CSR 2008)*, volume 5010 of *Lecture Notes in Computer Science*, pages 3–10, 2008.

[10] E. Allender and M. Koucký. Amplifying lower bounds by means of self-reducibility. In *IEEE Conference on Computational Complexity*, pages 31–40, 2008.

[11] S. Arora and B. Barak. *Computational Complexity: A Modern Approach.* Cambridge University Press. To appear, draft available at http://www.cs.princeton.edu/theory/complexity/.

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

[13] S. Arora and S. Safra. Probabilistic checking of proofs: A new characterization of NP. *J. ACM*, 45(1):70–122, 1998.

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

[15] L. Babai, L. Fortnow, N. Nisan, and A. Wigderson. BPP has subexponential time simulations unless EXPTIME has publishable proofs. *Computational Complexity*, 3:307–318, 1993.

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

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

[18] L. Berman and J. Hartmanis. On isomorphism and density of NP and other complete sets. *SIAM Journal on Computing*, 6:305–322, 1977.

[19] E. Bernstein and U. V. Vazirani. Quantum complexity theory. *SIAM J. Comput.*, 26(5), 1997.

[20] M. Blum. A machine-independent theory of the complexity of recursive functions. *J. ACM*, 14(2):322–336, 1967.

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

[22] G. Brassard. A note on the complexity of cryptography. *IEEE Transactions on Information Theory*, IT-25:232–233, 1979.

[23] H. Buhrman and J. M. Hitchcock. NP-hard sets are exponentially dense unless coNP $\subseteq$ NP/poly. In *IEEE Conference on Computational Complexity*, pages 1–7, 2008.

[24] J.-Y. Cai. $\mathrm{S}_2^p \subseteq \mathrm{ZPP}^{\mathrm{np}}$. *J. Comput. Syst. Sci.*, 73(1):25–35, 2007.

[25] V. T. Chakaravarthy and S. Roy. Oblivious symmetric alternation. In *Proc. Symposium on Theoretical Aspects of Computer Science (STACS)*, number 3884 in Lecture Notes in Computer Science, pages 230–241, 2006.

[26] T. Chow. Almost-natural proofs. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 2008.

[27] Clay Mathematics Institute. Millenium problems. http://www.claymath.org/millennium/.

[28] S. A. Cook. The complexity of theorem-proving procedures. In *Proc. ACM Symposium on Theory of Computing (STOC)*, pages 151–158, 1971.

[29] S. A. Cook. An overview of computational complexity. *Commun. ACM*, 26(6):400–408, 1983.

[30] S. A. Cook. The importance of the P versus NP question. *J. ACM*, 50(1):27–29, 2003.

[31] I. Dinur. The PCP theorem by gap amplification. *J. ACM*, 54(3):12, 2007.

[32] D.-Z. Du and K.-I. Ko. *Theory of Computational Complexity*. Wiley-Interscience, New York, 2000.

[33] U. Feige, S. Goldwasser, L. Lovász, S. Safra, and M. Szegedy. Interactive proofs and the hardness of approximating cliques. *J. ACM*, 43(2):268–292, 1996.

[34] M. R. Fellows and M. A. Langston. Nonconstructive tools for proving polynomial-time decidability. *J. ACM*, 35(3):727–739, 1988.

[35] L. Fortnow. Time-space tradeoffs for satisfiability. *J. Comput. Syst. Sci.*, 60:336–353, 2000.

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

[37] J. Friedman. Linear transformations in boolean complexity theory. In *Computation and Logic in the Real World (CiE 2007)*, volume 4497 of *Lecture Notes in Computer Science*, pages 307–315, 2007.

[38] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the theory of NP-completeness*. W.H. Freeman and Company, New York, 1979.

[39] M. R. Garey and D. S. Johnson. The complexity of near-optimal graph coloring. *J. ACM*, 23(1):43–49, 1976.

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

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

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

[43] P. Gopalan and V. Guruswami. Hardness amplification within NP against deterministic algorithms. In *IEEE Conference on Computational Complexity*, pages 19–30, 2008.

[44] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo. *Limits to parallel computation: P-completeness theory*. Oxford Univ. Press, 1995.

[45] J. Hartmanis. Gödel, von Neumann and the P=?NP problem. In G. Rozenberg and A. Salomaa, editors, *Current Trends in Theoretical Computer Science*, volume 40 of *World Scientific Series in Computer Science*, pages 445–450. World Scientific Press, 1993.

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

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

[48] J. Håstad. Some optimal inapproximability results. *J. ACM*, 48(4):798–859, 2001.

[49] A. Healy, S. P. Vadhan, and E. Viola. Using nondeterminism to amplify hardness. *SIAM J. Comput.*, 35(4):903–931, 2006.

[50] T. Holenstein. Parallel repetition: simplifications and the no-signaling case. In *Proc. ACM Symposium on Theory of Computing (STOC)*, pages 411–419, 2007.

[51] S. Homer. Minimal degrees for polynomial reducibilities. *J. ACM*, 34(2):480–491, 1987.

[52] S. Homer and A. Selman. *Computability and Complexity Theory*. Springer-Verlag, 2001.

[53] R. Impagliazzo. A personal view of average-case complexity. In *Structure in Complexity Theory Conference*, pages 134–147, 1995.

[54] R. Impagliazzo. Computational complexity since 1980. In *Proc. Conference on Foundations of Software Technology and Theoretical Computer Science (FST&TCS)*, volume 3821 of *Lecture Notes in Computer Science*, pages 19–47, 2005.

[55] R. Impagliazzo and A. Wigderson. $P = BPP$ if E requires exponential circuits: Derandomizing the XOR lemma. In *Proc. ACM Symposium on Theory of Computing (STOC)*, pages 220–229, 1997.

[56] D. S. Johnson. NP-completeness columns. Twenty-Six Columns published in *J. Algorithms* (1981-1992) and *ACM Trans. Algorithms* (2005 - present), available at http://www.research.att.com/ dsj/columns/.

[57] N. D. Jones. Space bounded reducibility among combinatorial problems. *J. Comput. Syst. Sci.*, 11:68–85, 1975.

[58] R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–104. New York, 1972.

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

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

[61] S. Kurtz, S. Mahaney, and J. Royer. The structure of complete degrees. In A. Selman, editor, *Complexity Theory Retrospective*, pages 108–146. Springer-Verlag, 1990.

[62] S. A. Kurtz, M. J. O'Donnell, and J. S. Royer. How to prove representation-independent independence results. *Inf. Process. Lett.*, 24(1):5–10, 1987.

[63] R. E. Ladner. On the structure of polynomial time reducibility. *J. ACM*, 22(1):155–171, 1975.

[64] L. Levin. Universal search problems. *Problems of Information Transmission*, 9:265–266, 1973.

[65] L. A. Levin. Average case complete problems. *SIAM J. Comput.*, 15(1):285–286, 1986.

[66] L. A. Levin. One-way functions and pseudorandom generators. *Combinatorica*, 7(4):357–363, 1987.

[67] L. A. Levin. Computational complexity of functions. *Theor. Comput. Sci.*, 157(2):267–271, 1996.

[68] C. Lund, L. Fortnow, H. Karloff, and N. Nisan. Algebraic methods for interactive proof systems. *J. ACM*, 39:859–868, 1992.

[69] S. Mahaney. Sparse complete sets for NP: Solution of a conjecture of Berman and Hartmanis. *J. Comput. Syst. Sci.*, 25(2):130–143, 1982.

[70] S. Mahaney. The isomorphism conjecture and sparse sets. In J. Hartmanis, editor, *Computational Complexity Theory*, pages 18–46. American Mathematical Society Proceedings of Symposia in Applied Mathematics #38, 1989.

[71] Merriam-Webster. *Webster's Seventh New Collegiate Dictionary*. Merriam-Webster, 1969.

[72] A. R. Meyer and L. J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 125–129, 1972.

[73] P. B. Miltersen and N. V. Vinodchandran. Derandomizing Arthur-Merlin games using hitting sets. *Computational Complexity*, 14(3):256–279, 2005.

[74] K. Mulmuley and M. A. Sohoni. Geometric complexity theory I: An approach to the P vs. NP and related problems. *SIAM J. Comput.*, 31(2):496–526, 2001.

[75] N. Nisan and A. Wigderson. Hardness vs. randomness. *J. Comput. Syst. Sci.*, 49:149–167, 1994.

[76] R. O'Donnell. Hardness amplification within NP. *J. Comput. Syst. Sci.*, 69(1):68–94, 2004.

[77] M. Ogiwara and O. Watanabe. On polynomial-time bounded truth-table reducibility of NP sets to sparse sets. *SIAM J. Comput.*, 20(3):471–483, 1991.

[78] C. H. Papadimitriou. NP-completeness: A retrospective. In *International Conference on Automata, Languages, and Programming (ICALP)*, volume 1256 of *Lecture Notes in Computer Science*, pages 2–6. Springer-Verlag, 1997.

[79] A. Razborov and S. Rudich. Natural proofs. *J. Comput. Syst. Sci.*, 55:24–35, 1997.

[80] R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems (reprint). *Commun. ACM*, 26(1):96–99, 1983.

[81] J. M. Robson. N by N checkers is exptime complete. *SIAM J. Comput.*, 13(2):252–267, 1984.

[82] R. Shaltiel and C. Umans. Simple extractors for all min-entropies and a new pseudorandom generator. *J. ACM*, 52(2):172–216, 2005.

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

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

[85] P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, 1997.

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

[87] C. F. Slot and P. van Emde Boas. On tape versus core; an application of space efficient perfect hash functions to the invariance of space. In *Proc. ACM Symposium on Theory of Computing (STOC)*, pages 391–400, 1984.

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

[89] L. J. Stockmeyer and A. R. Meyer. Cosmological lower bound on the circuit complexity of a small problem in logic. *J. ACM*, 49(6):753–784, 2002.

[90] L. Trevisan. On uniform amplification of hardness in NP. In *Proc. ACM Symposium on Theory of Computing (STOC)*, pages 31–38, 2005.

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

[92] D. van Melkebeek. A survey of lower bounds for satisfiability and related problems. *Foundations and Trends in Theoretical Computer Science*, 2:197–303, 2007.

[93] H. Vollmer. *Introduction to Circuit Complexity*. Springer, 1999.

[94] K. W. Wagner. More complicated questions about maxima and minima, and some closures of NP. *Theor. Comput. Sci.*, 51:53–80, 1987.

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

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

[97] I. Wegener. *Complexity Theory: Exploring the Limits of Efficient Algorithms*. Springer-Verlag, 2005.

[98] A. Wigderson. P, NP and mathematics - a computational complexity perspective. *Proceedings of the ICM 2006*, 1:665–712, 2007.

[99] R. Williams. Time-space tradeoffs for counting NP solutions modulo integers. In *IEEE Conference on Computational Complexity*, pages 70–82, 2007.

[100] P. Young. Juris Hartmanis: Fundamental contributions to isomorphism problems. In A. Selman, editor, *Complexity Theory Retrospective*, pages 28–58. Springer-Verlag, 1990.