On the Work of Madhu Sudan

Avi Wigderson

Madhu Sudan is the recipient of the 2002 Nevanlinna Prize. Sudan has made fundamental contributions to two major areas of research, the connections between them, and their applications.

The first area is coding theory. Established by Shannon and Hamming over fifty years ago, it is the mathematical study of the possibility of, and the limits on, reliable communication over noisy media. The second area is probabilistically checkable proofs (PCPs). By contrast, it is only ten years old. It studies the minimal resources required for probabilistic verification of standard mathematical proofs.

My plan is to briefly introduce these areas, their motivation, and foundational questions and then to explain Sudan's main contributions to each. Before we get to the specific works of Madhu Sudan, let us start with a couple of comments that will set up the context of his work.

- Madhu Sudan works in computational complexity theory. This research discipline attempts to rigorously define and study *efficient* versions of objects and notions arising in computational settings. This focus on efficiency is of course natural when studying computation itself, but it also has proved extremely fruitful in studying other fundamental notions such as proof, randomness, knowledge, and more. Here I will try to explain how the efficiency "eyeglasses" were key in this study of the notions proof (again) and error correction.
- Theoretical computer science is an extremely interactive and collaborative community. Sudan's work was not done in a vacuum, and much of the background to it, conceptual and technical, was developed by other people. The space I have does not allow me to give proper credit to all these people. A much better job has been done by Sudan himself; his homepage (`http://theory.lcs.mit.edu/~madhu/`) contains several surveys of these areas which give proper historical accounts and references. In particular see [13] for a survey on PCPs and [15] for a survey on the work on error correction.

Avi Wigderson is professor of mathematics at the Institute for Advanced Study, Princeton, and The Hebrew University, Jerusalem. His email address is `avi@ias.edu`.

Probabilistic Checking of Proofs

One informal variant of the celebrated $\mathcal{P}$ versus $\mathcal{NP}$ question asks, Can mathematicians, past and future, be replaced by an efficient computer program? We first define these notions and then explain the PCP theorem and its impact on this foundational question.

Efficient Computation

Throughout, by an *efficient* algorithm (or program, machine, or procedure) we mean an algorithm which runs at most some fixed polynomial time[1] in the *length* of its input. The input is always a finite

[1] *Time refers to the number of elementary steps taken by the algorithm. The choice of "polynomial" to represent efficiency is both small enough to often imply practicality and large enough to make the definition independent of particular aspects of the model, e.g., the choice of allowed "elementary operations".*

string of symbols from a fixed, finite alphabet. Note that an algorithm is an object of fixed size which is supposed to solve a problem on all inputs of all (finite) lengths. A problem is *efficiently computable* if it can be solved by an efficient algorithm.

Definition 1. *The class $\mathcal{P}$ is the class of all problems solvable by efficient algorithms.*

For example, the problems Integer Multiplication, Determinant, Linear Programming, Univariate Polynomial Factorization, and (recently established) Testing Primality are in $\mathcal{P}$.

Let us restrict attention (for a while) to algorithms whose output is always "accept" or "reject". Such an algorithm A solves a *decision* problem. The set L of inputs which are accepted by A is called the *language* recognized by A. Statements of the form "$x \in L$" are correctly classified as "true" or "false" by the efficient algorithm A, deterministically (and without any "outside help").

Efficient Verification

In contrast, allowing an efficient algorithm to use "outside help" (a guess or an alleged proof) naturally defines a proof system. We say that a language L is *efficiently verifiable* if there is an efficient algorithm V (for "Verifier") and a fixed polynomial p for which the following completeness and soundness conditions hold:

- For every $x \in L$ there exists a string π of length $|\pi| \leq p(|x|)$ such that V accepts the joint input (x, π).
- For every $x \notin L$, for every string π of length $|\pi| \leq p(|x|)$, V rejects the joint input (x, π).

Naturally, we can view all strings x in L as theorems of the proof system V. Those strings π which cause V to accept x are legitimate proofs of the theorem $x \in L$ in this system.

Definition 2. *The class $\mathcal{NP}$ is the class of all languages that are efficiently verifiable.*

It is clear that $\mathcal{P} \subseteq \mathcal{NP}$. Are they equal? This is the "$\mathcal{P}$ versus $\mathcal{NP}$" question [5], one of the most important open scientific problems today. Not only mathematicians but scientists and engineers as well daily attempt to perform tasks (create theories and designs) whose success can hopefully be efficiently verified. Reflect on the practical and philosophical impact of a positive answer to the question: if $\mathcal{P} = \mathcal{NP}$, then much of their (creative!) work can be performed efficiently by one computer program.

Many important computational problems, like the Travelling Salesman, Integer Programming, Map Coloring, Systems of Quadratic Equations, and Integer Factorization are (when properly coded as languages) in $\mathcal{NP}$.

We stress two aspects of efficient verification. The purported "proof" π for the statement "$x \in L$" must be short, and the verification procedure must be efficient. It is important to note that all standard (logical) proof systems used in mathematics conform to the second restriction: since only "local inferences" are made from "easily recognizable" axioms, verification is always efficient in the total length of statement and proof. The first restriction, on the length of the proof, is natural, since we want the verification to be efficient in terms of the length of the statement.

An excellent, albeit informal, example is the language MATH of all mathematical statements, whose proof verification is defined by the well-known efficient (anonymous) algorithm REFEREE.[2] As humans we are simply not interested in theorems whose proofs take, say, longer than our lifetime (or the three-month deadline given by EDITOR) to read and verify.

But is this notion of efficient verification—reading through the statement and proof, and checking that every new lemma indeed follows from previous ones (and known results)—the best we can hope for? Certainly as referees we would love some shortcuts as long as they do not change our notion of mathematical truth too much. Are there such shortcuts?

Efficient Probabilistic Verification

A major paradigm in computational complexity is allowing algorithms to flip coins. We postulate access to a supply of independent unbiased random variables which the *probabilistic* (or randomized) algorithm can use in its computation on a given input. We comment that the very rich theories (which we have no room to discuss) of pseudorandomness and of weak random sources attempt to bridge the gap between this postulate and "real-life" generation of random bits in computer programs.

The notion of efficiency remains the same: probabilistic algorithms can make only a polynomial number of steps in the input length. However, the output becomes a random variable. We demand that the probability of error, on *every* input, never exceed a given small bound ϵ. (Note that ϵ can be taken to be, e.g., $1/3$, since repeating the algorithm with fresh independent randomness and taking majority vote of the answers can decrease the error exponentially in the number of repetitions.)

Returning to proof systems, we now allow the verifier V to be a probabilistic algorithm. As above, we allow it to err (namely, accept false "proofs") with extremely small probability. The gain would be extreme efficiency: the verifier will access only a *constant* number of symbols in the alleged proof. Naturally, the positions of the viewed symbols can

[2] *This system can, of course, be formalized. However, it is better to have the* social *process of mathematics in mind before we plunge into the notions of the next subsection.*

be randomly chosen. What kind of theorems can be proved in the resulting proof system? First, let us formalize it.

We say that a language L has a *probabilistically checkable proof* if there is an efficient probabilistic algorithm V, a fixed polynomial p, and a fixed constant c for which the following completeness and probabilistic soundness conditions hold.

- For every $x \in L$ there exists a string π, of length $|\pi| \leq p(|x|)$, such that V accepts the joint input (x, π) with probability 1.
- For every $x \notin L$, for every string π of length $|\pi| \leq p(|x|)$, V rejects the joint input (x, π) with probability $\geq 1/2$.
- On every input (x, π), V can access at most c bits of π.

Note again that executing the verifier independently a constant number of times will reduce the "soundness" error to an arbitrarily small constant without changing the definition. Also note that randomness in the verifier is essential if it probes only a constant (or even logarithmic) number of symbols in the proof. The reader may verify that such deterministic verification can exist only for easy languages in $\mathcal{P}$.

Definition 3. *The class $\mathcal{PCP}$ is the class of all languages that have a probabilistically checkable proof.*

The main contribution of Sudan and his colleagues to this area is one of the deepest and most important achievements of theoretical computer science.

Theorem 1. (The PCP Theorem [2, 1]). $\mathcal{PCP} = \mathcal{NP}$.

In words, *every* theorem that can be efficiently verified can also be verified probabilistically by viewing only a *fixed number* of bits of the purported proof. If the proof is correct, it will *always* be accepted. If it is wrong (in particular, when the input statement is not a theorem, all "proofs" will be wrong), it will be rejected with high probability despite the fact that the verifier hardly glanced at it.

The proof of the PCP theorem, from a very high-level point of view, takes a standard proof system (a problem in $\mathcal{NP}$) and constructs from it a very robust proof system. A correct proof in the former is transformed to a correct proof in the latter. A false "proof" in the former (even if it has only one bug in a remote lemma, to use the refereeing metaphor again) is transformed to a "proof" littered with bugs, so many that a random sample of a few bits would find one.

This conversion appears related to error-correcting coding (which is our second topic), and indeed it is. However, it is quite a bit more, as the encoded string has a "meaning": it is supposed to be a proof of a given statement, and the coding must keep it so.

The conversion above is efficient and deterministic. So in principle an efficient program can be written to convert standard mathematical proofs to robust ones which can be refereed in a jiffy.

The PCP theorem challenges the classical belief that proofs have to be read and verified fully for one to be confident of the validity of the theorem. Of course one does not expect the PCP theorem to dramatically alter the process of writing and verifying proofs (any more than one would expect automated verifiers of proof systems to replace the REFEREE for journal papers). In this sense the PCP theorem is just a statement of philosophical importance. However, the PCP theorem does have significant implications of immediate relevance to the theory of computation, and we explain this next.

Hardness of Approximation

The first and foremost contribution thus far to understanding the mystery of the $\mathcal{P}$ versus $\mathcal{NP}$ question was the discovery of $\mathcal{NP}$-completeness and its ubiquity by Cook, Levin, and Karp in the early 1970s.

Roughly speaking, a language is $\mathcal{NP}$-complete if it is the hardest in the class $\mathcal{NP}$. More precisely, a language L is $\mathcal{NP}$-complete if *any* efficient algorithm for it can be used to efficiently solve every other language in $\mathcal{NP}$. Note that by definition every $\mathcal{NP}$-complete language is as hard to compute as any other. Moreover, $\mathcal{P} = \mathcal{NP}$ if and only if any $\mathcal{NP}$-complete language is easy, and $\mathcal{P} \neq \mathcal{NP}$ if and only if any $\mathcal{NP}$-complete language is hard.

As it turns out, almost every language known in $\mathcal{NP}$ is known to be either $\mathcal{NP}$-complete or in $\mathcal{P}$. The great practical importance of the $\mathcal{P}$ versus $\mathcal{NP}$ question stems from the fact that numerous outstanding problems in science and engineering turn out to be $\mathcal{NP}$-complete. For computer programmers or engineers required by their boss to find an efficient solution to a given problem, proving it $\mathcal{NP}$-complete is the ultimate excuse for not doing it; after all, it is as hard as all these thousands of other problems which scientists in various disciplines have attempted unsuccessfully.

Knowing the practical world, we suspect that the boss would not be impressed. In real life we need to solve impossible problems too. To do that, we reduce our expectations! An almost universal situation of this type is some optimization problem for which finding the *optimal solution* is $\mathcal{NP}$-complete or harder. In this situation the boss would ask for an efficient algorithm for some "reasonable" *approximation* of the optimal solution. Many success stories exist; an important example is the efficient algorithm for approximating (by any constant factor > 1) the volume of a convex body of high dimension. What about failure? Does the theory of $\mathcal{NP}$-completeness provide any excuses to our poor programmers if they fail again?

For twenty years there was essentially no answer to this question. The complexity of approximation problems was far from understood. It was clear that this area is much richer/murkier than that of decision problems. For illustration, consider the following three optimization problems.

- **Linear Equations:** Given a system of n linear equations, say over the finite field $GF(2)$, determine the maximal number that can be satisfied simultaneously.
- **Set Cover:** Given a collection of subsets of a given finite universe of size n, determine the size of the smallest subcollection that covers every element in the universe.
- **Clique:** Given a finite graph on n vertices, find the size of the largest subset of vertices which are pairwise connected by edges.

For each of these problems, finding the optimal solution is $\mathcal{NP}$-complete. Some naive approximation algorithms have existed for a long time, and no one could improve them. They yield completely different approximation factors.

- **Linear Equations:** A random assignment will satisfy on average half the equations, so it is at most a factor 2 from optimal. Try beating it.
- **Set Cover:** A simply greedy algorithm, collecting subsets so that the next one covers as many yet uncovered elements as possible, will be a factor $\ln n$ from optimal. Try proving it.
- **Clique:** A trivial solution is a 1-vertex clique, which is within a factor n of optimal. Somewhat more elaborate algorithms give a factor $n/(\log n)^2$. Think it pathetic? Try improving it to (the still pathetic?) $n^{0.999}$.

The PCP theorem, as well as many other technical developments of Sudan and other researchers, has paved the way to an almost complete understanding of how well we can approximate different natural optimization problems efficiently. These developments vary PCPs in many ways and study many other "resources" of the proof verification process, beyond the number of queries to the proof and the error probability. In particular, as the following three different theorems show, the trivial approximation algorithms above are essentially the best possible.

- **Linear Equations:** Approximation by a factor of $2-\epsilon$ is $\mathcal{NP}$-hard[3] for every $\epsilon > 0$ [11].
- **Set Cover:** Approximation by a factor of $(1-\epsilon)\ln n$ is $\mathcal{NP}$-hard for every $\epsilon > 0$ [6].
- **Clique:** Approximation by factor of $n^{1-\epsilon}$ is $\mathcal{NP}$-hard for every $\epsilon > 0$ [10].

[3] *We use "hard" rather than "complete", as these problems are not languages in $\mathcal{NP}$ as defined above (but they can be so defined). The meaning remains: an efficient algorithm for the problem would yield $\mathcal{P} = \mathcal{NP}$.*

The connection between PCPs and the hardness of approximation was established in [7]. The basic idea is the following: The PCP theorem provides a natural optimization problem which cannot be efficiently approximated by any factor better than 2. Namely, fix a verifier V (and thus a language L it accepts, as in the PCP theorem). Given x, find the maximum acceptance probability of V on x by any proof π. Clearly, by the definition of probabilistic verification, beating a factor of 2 efficiently means distinguishing between those $x \in L$ and those $x \notin L$. By the theorem, L can be any problem in $\mathcal{NP}$, so such an efficient approximator would yield $\mathcal{P} = \mathcal{NP}$.

This optimization problem serves the same purpose that the satisfiability of Boolean formulae served when discovered as the first $\mathcal{NP}$-complete language. From then on efficient reductions, namely, transformations of one problem to another, could be used to prove completeness. Here, too, reductions are used to get the above theorems on hardness of approximation. However, these reductions are far more intricate than for decision problems; the difference in approximability of these different problems is but the first indication of the richness and complexity of this area.

List Decodable and Implicit Error-Correcting Codes

Unique Decoding

You have some precious information, which you may want to store or communicate. It is represented, say, by K symbols from some fixed alphabet Σ. To protect any part of it from being lost, you are prepared to store/communicate it with redundancy, using N symbols of the same alphabet. Then it will be subject to (a process we view as) an adversary who may destroy or change, say, T of the symbols in the encoding. A decoding process must then be able to recover the original information.

This scenario is the core of a multitude of practical devices you use—CDs, satellite transmissions, Internet communications, and many others. The problem of determining the best achievable relationships between the parameters K, N, and T was raised by Hamming about fifty years ago. In a slightly different scenario, when changes are random, it was raised even earlier by Shannon. The variety of related models and problems constitute the large and active field of coding theory and its close relative, information theory.

Once again, efficiency requirements enrich the problems tremendously. It was only a few years ago that optimal *codes* (having linear-time encoding and decoding algorithms) were developed. These nearly match Shannon's completely nonconstructive bounds and apply as well to the Hamming problem (on which we focus from now on for simplicity).

One central feature of this huge body of work was its focus on *unique decoding*: you want to recover the original information when the corrupted codeword defines it unambiguously. This requires that the encoding of *any* two information words differ in at least $2T+1$ positions.

Is there a meaning to useful decoding when the Hamming distance (the number of differing symbols) is less than $2T+1$ and ambiguity is unavoidable? Can one achieve it efficiently? Why bother?

List Decoding

The questions above were pondered in the past decades. It was realized that ambiguous decoding would be useful if the decoding process generated a short *list* of candidates, one of which is the original information. Moreover, it was realized that *in principle* such a short list exists even if distances between pairs of encodings are close to T. This answers the "why bother?" question immediately: this would drastically improve the ratio between K and N (in this area constant factor improvements are "drastic"). But no nontrivial code was known to support such decoding even remotely efficiently.

Sudan's extremely elegant algorithm to list-decode Reed-Solomon codes completely changed the situation. It started a snowball rolling which transformed large areas in this field in a matter of two years, again led by Sudan and his colleagues. Moreover, once discovered, these codes were applied to solve theoretical problems, mainly within complexity theory, providing completely different answers (that we mention later) to the "why bother?" question.

The Reed-Solomon codes, the old result about unique decoding, and Sudan's result on list-decoding should appeal to any mathematician, whether interested or not in error correction. Here is the setup; the reader can easily relate the parameters below to those above.

Fix a finite field F and an integer $d \leq |F|$. My information is a degree d polynomial p over F, and I encode it simply as a table of the values of p on all elements of F. Now suppose an adversary changes the table, the only restriction being to leave at least t of the $|F|$ positions in agreement with p. Can p be recovered from the table efficiently?

For unique decoding, namely, for the table to uniquely define one polynomial, clearly we need $t > |F|/2$ (otherwise we could fill half the table with the values of one polynomial, p_1, and the other half with values of a different polynomial, p_2). When $d < |F|/2$, this bound turns out to be not only necessary but also sufficient!

Theorem 2 [12]. *There is an efficient algorithm that recovers p from any table that agrees with p on $t > |F|/2$ elements of F.*

Note that the decoding problem is a nonlinear one and brute-force search takes time about $|F|^d$, which is prohibitive when d is large. The algorithm above, polynomial in $|F|$ and d, uses (efficient) univariate polynomial factoring over F. Sudan's algorithm uses factorization of bivariate polynomials to list-decode even if the fraction of agreement goes to zero with $|F|$!

Theorem 3 [14, 9]. *There is an efficient algorithm which, for every $\epsilon > \sqrt{d/|F|}$, recovers a list of $O(1/\epsilon^2)$ polynomials containing p from any table that agrees with p on $t > \epsilon|F|$ elements of F.*

Put differently, given *any* function g from F to itself, this algorithm *efficiently* recovers *all* degree d polynomials which agree with g on $\epsilon|F|$ arguments.

Implicit Codes

So we have codes (for unique and list decoding) that achieve near-optimal parameters in wide ranges and whose encoding and decoding takes linear time—essentially the time to read the input and write down the output. What more can we want? The answer is, try to use sublinear time: some root of the input length; better, a logarithm of it; or, even better, constant time, independent of the input length!

If you consider the amounts of data biologists and astronomers have to understand and explain or the amounts of data on the World Wide Web from which we want to gather some useful information, you will realize that sublinear algorithms which look at only a tiny fraction of the data are in vast demand. If you consider the way we search through a phone book or gather simple statistics via sampling, you will realize that at least for very simple tasks such algorithms exist. The importance of such algorithms has made this area grow in recent years, with many more examples of sophisticated algorithms for far less trivial tasks.

Computational complexity suggests a natural way to represent huge objects. A basic object is a function, and a basic representation for it is a program that computes it. Returning to the coding problem, think of K (and hence of N) as being so large that we have neither room nor time to write it down explicitly. The K information symbols we wish to encode are given by a program P which on input at position $i < K$ gives the value of the ith information symbol. Note that such a program can be extremely succinct: it can have size exponentially smaller than K.

The encoder will take P and produce another program Q of the same type; on input $j < N$ it will output the jth symbol in the encoding. Now assume an adversary changes Q to yield a Q' that agrees with Q only on some small fraction of the inputs. Can we recover P in any reasonable sense? The appropriate sense suggests itself: we should construct a small program P' that for any input $i < K$ may query Q' in a few places and then output the correct ith information symbol. In the list-decoding case, we may provide a few such programs, one of which does the job *for every i*.

Amazingly enough, this can be done! Sudan and his colleagues have provided a strong positive answer to this challenge. Stated informally, what they have done is to construct codes in which such implicit decoding can be done very efficiently. The new programs use very few queries to the corrupted program Q', even if it agrees with Q only on a vanishingly small fraction of the domain. (See [16] for full definitions and statements of results.)

A moment's reflection will convince you that in this setting the recovered programs have to be probabilistic, as otherwise the adversary would know in advance which values of Q' will be queried and will change only them. However, as usual, correct output is guaranteed with arbitrarily high probability.

Applications, Connections, and Techniques

The applications of these results far exceed the domain of error correction. Let me mention a few directly affected areas, without getting into details.

- Program Testing [4]: The ability to design robust programs whose correctness on single inputs can be automatically verified.
- Hardness Amplification [16]: The ability to efficiently convert functions which are hard on rare inputs to functions which are hard on random inputs. Such conversions are important to cryptography and to pseudorandom number generation.
- Computational Learning Theory [8]: Some of the results above can be interpreted as learning concepts from a few examples.
- Probabilistic Proof Systems [3]: The reader may have noticed that the probabilistic verifier in PCPs is a prime example of an algorithm which looks at only a tiny part of its input. These implicit and list-decodable codes play an essential role in the design of PCPs, as well as in other important proof systems not mentioned here.

To conclude, we mention that these topics illuminate the power of algebra in computer science. Many of the constructions use algebraic objects (and theorems about them). The efficient use of these objects often calls for sophisticated algorithms to manipulate them. Again, this has been a major endeavor on the boundary between algebra and computer science, with many other applications and with cross-fertilization in both directions. A look at his home page (`http://theory.lcs.mit.edu/~madhu/`) reveals not only the extent to which Sudan was part of this research but also his investment in collecting, clarifying, and conveying this knowledge in teaching and writing.

References

[1] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy, Proof verification and the hardness of approximation problems, *J. ACM*, **45** (1998), 501–55.

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

[3] Sanjeev Arora and Madhu Sudan, Improved low-degree testing and its applications, *Proc. Twenty-Ninth Annual ACM Sympos. Theory Comput.* (El Paso, Texas), ACM, New York, 1999, pp. 485–95.

[4] Manuel Blum, Michael Luby, and Ronitt Rubinfeld, Self-testing/correcting with applications to numerical problems, *J. Comput. System Sci.* **47** (1993), 549–95.

[5] Clay Mathematical Institute, `http://www.claymath.org/prizeproblems/pvsnp.htm`.

[6] Uriel Feige, A threshold of $\ln n$ for approximating set cover, *J. ACM* **45** (1998), 634–52.

[7] Uriel Feige, Shafi Goldwasser, Lászlo Lóvász, Shmuel Safra, and Mario Szegedy, Interactive proofs and the hardness of approximating cliques, *J. ACM* **43** (1996), 268–92.

[8] Oded Goldreich, Ronitt Rubinfeld, and Madhu Sudan, Learning polynomials with queries: The highly noisy case, *SIAM J. Discrete Math.* **13** (2000), 535–70.

[9] Venkatesan Guruswami and Madhu Sudan, Improved decoding of Reed-Solomon and algebraic-geometric codes, *IEEE Trans. Inform. Theory* **45** (1999), 1757–67.

[10] Johan Håstad, Clique is hard to approximate within $n^{1-\epsilon}$, *Acta Math.* **182** (1999), 105–42.

[11] ——, Some optimal inapproximability results, *J. ACM* **48** (2001), 798–859.

[12] W. Wesley Peterson, Encoding and error-correction procedures for Bose-Chaudhuri codes, *IEEE Trans. Inform. Theory* **6** (1960), 459–70.

[13] Madhu Sudan, Probabilistically checkable proofs. Scribed notes by Venkatesan Guruswami, to appear in *Lecture Notes of IAS/Park City Summer School on Complexity Theory*.

[14] ——, Decoding of Reed-Solomon codes beyond the error-correction bound, *J. Complexity* **13** (1997), 180–93.

[15] ——, List decoding: Algorithms and applications, *SIGACT News* **31** (2000), 16–27.

[16] Madhu Sudan, Luca Trevisan, and Salil Vadhan, Pseudorandom generators without the XOR lemma, *Proc. 31st Annual ACM Sympos. Theory Comput.* (1999), 537–46.