DIMACS Technical Report 95-22
July 1995

The Use of Coding Theory in Computational Complexity

by

Joan Feigenbaum[1]
AT&T Bell Labs, Room 2C-473
600 Mountain Avenue
Murray Hill, NJ 07974
jf@research.att.com

[1]Permanent Member of DIMACS

DIMACS is a cooperative project of Rutgers University, Princeton University, AT&T Bell Laboratories and Bellcore.

DIMACS is an NSF Science and Technology Center, funded under contract STC–91–19999; and also receives support from the New Jersey Commission on Science and Technology.

ABSTRACT

The interplay of coding theory and computational complexity theory is a rich source of results and problems. This article surveys three of the major themes in this area:

- the use of codes to improve algorithmic efficiency
- the theory of program testing and correcting, which is a complexity theoretic analogue of error detection and correction
- the use of codes to obtain characterizations of traditional complexity classes such as NP and PSPACE; these new characterizations are in turn used to show that certain combinatorial optimization problems are as hard to approximate closely as they are to solve exactly.

These results were presented at a short course on Coding Theory at the American Mathematical Society winter meeting in San Francisco, January 1995.

1 Introduction

Complexity theory is the study of efficient computation. Faced with a computational problem that can be modelled formally, a complexity theorist seeks first to find a solution that is provably efficient and, if such a solution is not found, to prove that none exists. Coding theory, which provides techniques for "robust representation" of information, is valuable both in designing efficient solutions and in proving that efficient solutions do not exist.

This article surveys the use of codes in complexity theory. The improved upper bounds obtained with coding techniques include bounds on the number of random bits used by probabilistic algorithms and the communication complexity of cryptographic protocols. In these constructions, codes are used to design small sample spaces that approximate the behavior of large sample spaces. Codes are also used to prove lower bounds on the complexity of approximating numerous combinatorial optimization functions. Finally, coding theory plays an important role in new characterizations of traditional complexity classes, such as NP and PSPACE, and in an emerging theory of program testing.

Section 2 reviews the concepts from both complexity theory and coding theory that we will need in the following sections. Section 3 demonstrates how codes are used to prove complexity theoretic upper bounds. The theory of program testing and correcting is presented in Section 4. Lower bound applications and new characterizations of standard complexity classes are presented in Section 5.

2 Preliminaries

This section briefly reviews the basic elements of the two discplines under consideration, i.e., complexity theory and coding theory. If more than a brief review is needed, refer to one of the standard textbooks [27, 40, 39, 43].

2.1 Complexity Theory

We assume that the reader is familiar with the Turing machine (TM) model of computation. (A formal definition of this model can be found in any introductory book on computational complexity, e.g., Garey and Johnson [27] or Papadimitriou [43].) However, most of the definitions presented in this section require only an intuitive understanding of the notions of "time complexity" or "space complexity" of a computational problem; readers unfamiliar with the definition of a TM can understand our main points in terms of the number of operations performed or the number of memory locations used by a standard computer program.

In general, a function we are interested in computing is defined on some infinite domain I. There is a sequence of subdomains I_1, I_2, ..., such that $I = \cup_{n \geq 1} I_n$ and a constant c such that, for all $x \in I_n$, $n^{1/c} \leq |x| \leq n^c$, where $|x|$ is the number of bits in some suitable binary encoding of x. For example, we may have $I = \{0,1\}^+$ and $I_n = \{0,1\}^n$, in which case we can take $c = 1$. If p_n is an n-bit prime, then I_n may consist of n-tuples of elements in $\mathrm{GF}(p_n)$, in which case we can take $c = 2$. We refer to elements of I_n as *inputs of size n*. So "size" is a generalization of binary length.

A *polynomial-time Turing machine* is a TM for which there is a specific polynomial, say $p(n)$, that bounds the running time; that is, on inputs of size n, the machine always halts after at most $p(n)$ steps. We denote by FP the class of functions computable by polynomial-time TMs. Note that a TM that computes a boolean function can also be interpreted as recognizing a language. For example, "computing f," where $f(x) = 1$ if x encodes a planar graph and $f(x) = 0$ otherwise, is the same as "recognizing the language of planar graphs." In general, the deterministic Turing machine M is said to recognize the language $\{x \mid f(x) = 1\}$, which is denoted $L(M)$. Thus, FP is a generalization of the complexity class P of polynomial-time recognizable languages. *Polynomial-space Turing machines* and the complexity classes FPSPACE and PSPACE are defined in exactly the same manner as polynomial-time TMs, FP, and P, except that the computational resource that is bounded by the polynomial $p(n)$ is space instead of time. *Exponential-time Turing machines* and the complexity classes FEXP and EXP are defined analogously to polynomial-time TMs, FP, and P, except that all occurrences of "polynomial" and "$p(n)$" are replaced by "exponential" and "$2^{p(n)}$."

A *nondeterministic polynomial-time Turing machine* is a polynomial-time TM that, at each step in the computation, has several choices for its next move. The set of all possible computations of such a machine M on input $x \in I_n$ can be viewed as a tree. Because the total running time of M is bounded by $p(n)$, the paths from the root to the leaves of the computation tree may all be taken to have length exactly $p(n)$; short computation paths may be padded with "dummy steps" if necessary. Each leaf in the computation tree is labelled with the output that M produces on input x if, during the computation, it makes the sequence of choices represented by the path to this leaf. If all of the labels are 0s and 1s, then we can define the language $L(M)$ accepted by M. The input x is in $L(M)$ if there is at least one leaf in the computation tree that is labelled 1. The path to such a leaf is called an *accepting computation of M on input x* or a *witness that $x \in L(M)$*. The set of all languages accepted by nondeterministic polynomial-time TMs constitutes the complexity class NP.

NP plays a prominent role in computational complexity. Indeed the central unsolved problem in complexity theory is the P = NP? question. The main reason for the centrality of the complexity class NP is the large number of natural computational problems that can be formalized as NP languages. Examples include:

- Satisfiability of propositional formulae: The NP language SAT consists of strings that encode satisfiable propositional formulae in conjunctive normal form (CNF). For example, $(x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2)$ is an element of the language SAT, because the assignment $x_1 = \mathrm{T}$, x_2=T satisfies both clauses. On the other hand, $(x_1 \vee x_2) \wedge (x_1 \vee \overline{x_2}) \wedge \overline{x}_1$ is not in SAT, because none of the four truth assignments to the variables x_1 and x_2 satisfies all three clauses simultaneously. In most settings, it suffices to consider the language 3SAT, in which the formulae are conjuncts of clauses that are themselves disjuncts of three literals.

- Colorability of graphs: The NP language COLOR consists of pairs (G, k) such that the graph G is k-colorable. That is, (G, k) is in COLOR if and only if there is a mapping σ from $V(G)$ to $\{1, \ldots, k\}$ such that, if $\{v, w\} \in E(G)$, then $\sigma(v) \neq \sigma(w)$.
- Clique number of graphs: The NP language CLIQUE consists of pairs (G, k) such that the graph G has a k-clique. That is, (G, k) is in CLIQUE if and only if there is a subset $\{v_1, \ldots, v_k\}$ of $V(G)$ such that $\{v_i, v_j\} \in E(G)$, for $1 \leq i < j \leq k$.

These three NP languages, along with thousands of others that arise naturally in applications, are *NP-complete*. A language L is NP-complete if it is in NP and, for any other language L' in NP, there is a function R in FP (called a *polynomial-time reduction*) such that $x \in L'$ if and only if $R(x) \in L$. This means, for example, that if there were a polynomial-time decision algorithm for 3SAT, there would be a polynomial-time algorithm for every L in NP: To decide whether x is in L, compute $R(x)$ and test whether it is in 3SAT. Thus, the NP-complete languages are "as hard as any language in NP," and, barring an unexpected resolution of the P = NP? question in the affirmative, they cannot be decided by deterministic polynomial-time algorithms. See Garey and Johnson [27] for a comprehensive formal treatment of the theory of NP-completeness.

The concept of NP-completeness and the proof that SAT is NP-complete were put forth in the seminal paper of Cook [18]. Similar ideas were developed independently by Levin [36]. Shortly thereafter, Karp [34] established the fundamental importance of the concept by proving the NP-completeness of many languages that arise naturally in applications, including 3SAT, CLIQUE, and COLOR. In the ensuing years, thousands of NP-completeness results were proven. A standard question (along with a body of technique for answering it) was established in the computer science repetoire: When confronted with a combinatorial problem for which you cannot find a good algorithm, formulate it as a language-recognition problem, and try to prove that the language is NP-complete. Such problems are now divided into those for which polynomial-time algorithms are known (e.g., maximum matching and network flow), those that are NP-complete, and a few (e.g., graph isomorphism[1]) whose status is unknown. Before Cook and Karp established the concept of NP-completeness, there was no coherent explanation for the fact that certain problems (like matching) had efficient algorithms, while others that did not seem very different at first glance (like CLIQUE), did not. With the theory of NP-completeness in hand, such an explanation is easy: A polynomial-time algorithm for, say, CLIQUE would resolve the P = NP? question in the affirmative and hence is not something we expect to find.

It is often more natural to formalize a computational task as an NP "search problem" or an NP "optimization function," rather than as an NP language-recognition problem. The search problems corresponding to the languages defined above are:

- Given a CNF or 3CNF formula, find a truth assignment that satisfies as many of the clauses as possible.

[1] Although there is no *proof* that graph isomorphism is not NP-complete (nor is one expected any time soon, because such a proof would imply that P $\neq$ NP), there is some evidence that it is not – see Boppana, Hastad, and Zachos [15] or Schoening [50].

- Given a graph, color its vertices using as few colors as possible so that no two adjacent vertices are the same color.
- Find a maximum-sized clique in a given graph.

The corresponding optimization functions are:

- Given a CNF or 3CNF formula ϕ, what is the largest k such that k clauses of ϕ are simultaneously satisfiable by a truth assignment? (These are the optimization functions MAX-SAT and MAX-3SAT, respectively.)
- Given a graph, what is the smallest number of colors needed to color its vertices such that no two adjacent vertices are the same color? (This is the optimization function *chromatic number*.)
- Given a graph, what is number of vertices in its largest clique? (This is the optimization function *clique number*.)

These optimization functions are *NP-hard*, just as the languages on which they are based are NP-complete. This means that the task of deciding membership in any NP language can be reduced in polynomial time to the task of computing one of these optimization functions. Thus, none of these optimization functions is in FP, unless P = NP.

One natural question to ask is whether these optimization problems can be solved approximately in polynomial time. An *ϵ-approximation algorithm* for an NP optimization function f takes an instance x and outputs an estimate E such that $E/(1+\epsilon) \leq f(x) \leq E(1+\epsilon)$. For example, if there were a polynomial-time (1/2)-approximation algorithm for the chromatic number function χ (as there seems not to be – see [38]), it would take a graph G as input and output a number E such that $2E/3 \leq \chi(G) \leq 3E/2$. A *polynomial-time approximation scheme* (PTAS) for an NP optimization function f takes as input a parameter $\epsilon > 0$ and outputs an ϵ-approximation algorithm for f; the running time of this approximation algorithm is polynomial in the size of inputs to f, but it may depend superpolynomially on ϵ.

We say that *ϵ-approximating a function f is NP-hard* if the task of deciding membership in any NP language can be reduced in polynomial time to the task of ϵ-approximating f. Just as it is natural to try to prove that an optimization function is NP-hard if one has failed to find a polynomial-time algorithm for it, it is also natural to try to prove that ϵ-approximating this function is NP-hard if one has failed to find a polynomial-time ϵ-approximation algorithm for it. While general techniques for proving NP-hardness of exact computations have been known since the early 1970s (see [27]), none were known for proving NP-hardness of approximate computations until recently. Coding theory plays a surprising role in these new techniques, as explained in Section 5 below.

The reason that standard NP-completeness theory does not help in proving nonapproximability of NP optimization functions is that typical polynomial-time reductions between NP-complete language-recognition problems do not preserve approximability. That is, if

L_1 and L_2 are NP languages with associated optimization functions f_1 and f_2, and R is a polynomial-time reduction from L_1 to L_2, then an approximation to $f_2(R(x))$ is typically of no help in computing an approximation to $f_1(x)$. A major step forward in the development of a general theory of approximability was taken by Papadimitriou and Yannakakis [44] in their seminal paper on the complexity class MAX-SNP. Before the introduction of MAX-SNP, approximability of NP-hard optimization functions was in the same state that exact optimization was in before the establishment of NP-completeness: Good approximation methods could be found for some problems (such as bin packing) but not for others (such as chromatic number), and there was no coherent explanation of which was which. Papadimitriou and Yannakakis's paper established the framework in which to classify approximation problems in the same way that the early work of Cook and Karp established a framework for exact optimization problems.

We give a definition (due to Sudan [57]) of MAX-SNP that is convenient for the results discussed in Section 5; it is equivalent to the original definition given in [44]. A *constraint of arity* c is a mapping from c boolean variables to the range $\{0,1\}$. A constraint is *satisfied* by a truth assignment if it evaluates to 1 at that assignment. The *constraint satisfaction problem* (CSP) is a family of optimization functions parameterized by the arity c and is defined as follows. An instance of c-CSP is a set $\{C_1,\ldots,C_m\}$ of constraints of arity c on variables $x_1,\ldots,x_n$. The value of the function c-CSP on instance $\{C_1,\ldots,C_m\}$ is the largest integer k such that there is a truth assignment to $x_1,\ldots,x_n$ that simultaneously satisfies k of the input constraints.

An optimization function is in the complexity class MAX-SNP if and only if it can be expressed as c-CSP for some c. For example, it is easy to see that the function MAX-3SAT is in MAX-SNP; just let $c=3$ and the clauses in an input formula be the constraints. In fact, MAX-3SAT is complete for the class MAX-SNP with respect to *linear reductions* [44]. These are reductions that are "approximability-preserving," in the sense that, if MAX-SNP functions f and g are such that f is linearly reducible to g and g has a polynomial-time ϵ-approximation algorithm, then f has a polynomial-time $d\epsilon$-approximation algorithm, for some constant d; exactly which constant d is achievable depends on how efficiently the reduction preserves approximability. (See [1, 44, 57] for a precise definition of linear reductions and many more examples of problems that are complete for MAX-SNP.) Papadimitriou and Yannakakis [44] show that every function in MAX-SNP is ϵ-approximable for some constant ϵ. They pose the basic question "do MAX-SNP-complete functions have PTASs?" This is the central question in the theory of approximability of NP optimization functions, because the definitions are such that all MAX-SNP functions have PTASs if the complete ones do. The theory of probabilistically checkable proof systems (see Section 5), in which coding techniques play a crucial role, has essentially resolved this question in the negative: MAX-SNP functions do not have PTASs unless P = NP.

A *nondeterministic exponential-time Turing machine* is defined exactly the same way as a nondeterministic polynomial-time Turing machine, except that all occurrences of "polynomial" and "$p(n)$" are replaced by "exponential" and "$2^{p(n)}$." The languages accepted by nondeterministic exponential-time TMs constitute the complexity class NEXP.

A *probabilistic polynomial-time Turing machine* is a polynomial-time TM that has an extra "random tape" of 0s and 1s that can be read as necessary. The bits on this tape are the outcomes of independent, unbiased coin tosses. Let M be a probabilistic polynomial-time TM with running time $p(n)$. On any input $x \in I_n$, M tosses at most $p(n)$ coins (i.e., reads at most $p(n)$ bits from the random tape), because it always halts after at most $p(n)$ steps. If the output of M is a single bit, we require that, on any input $x \in I_n$, M either output 1 with probability at least 3/4 or output 0 with probability at least 3/4; this probability is computed over the uniform distribution on $\{0,1\}^{p(n)}$ (i.e., over the probability space of all coin-toss sequences that M may make on input x). The language L recognized by such a machine M, i.e., the inputs x on which M outputs 1 with probability at least 3/4, is said to be in the complexity class BPP. The probability of error can be reduced from 1/4 to $2^{-q(n)}$, for any polynomial q, at the expense of increasing the number of coin tosses and the running time by a corresponding polynomial factor. (The issue of how many random bits a BPP machine uses is revisited in Section 3.1 below.) Note that the definition of BPP allows for two-sided error: The TM may give a wrong answer both in the case that $x \in L$ and in the case that $x \notin L$. Probabilistic polynomial-time TMs that admit only one-sided error and other formal definitions of probabilistic computation are treated at length in, e.g., [31, 58].

An *oracle Turing machine* M has access to an auxiliary "oracle tape" from which it can get answers to computations that it may not be able to perform itself. The "base machine" M may be deterministic, nondeterministic, or probabilistic. The oracle may itself be a machine or program of any of these three types, or it may be a nonrecursive function, i.e., a function that is not computable by any TM with any amount of computational resources. When M asks for the value of the oracle on some string y, this is referred to as an "oracle call," and it costs M one unit of time. The reason that it makes sense to charge a machine M only one unit of time for information that may be very expensive (or even impossible) to compute is that it allows us to model formally the notion of separating a computational task into its constituent parts. We can encapsulate part of the task in the oracle O and then ask what else the base machine M has to do in order to "reduce" the computational task to a sequence of calls to O.

The notation $M(x)$ denotes the output of Turing machine M on input x. If M is a deterministic machine, then it defines a computable function, and $M(x)$ is just the value of this function at x. If M is a nondeterministic machine, then it defines a multivalued partial function. If $x \in L(M)$, then the partial function's value at x is the set of accepting computations that witness this fact; if $x \notin L(M)$, then the function is undefined at x. If M is a probabilistic machine, then $M(x)$ is a random variable. We use $M^O(x)$ to denote the output of oracle Turing machine M on input x when run with oracle O. If O is a probabilistic oracle, then the output distribution is determined by the algorithms and the coin-toss sequences of both M and O.

The shorthand $m = poly(n)$ means that there is a polynomial p such that, for all sufficiently large n, $m \leq p(n)$. It is used when the existence of such a polynomial is what is important, rather than the specific polynomial p.

2.2 Coding Theory

We now review a small number of definitions from basic coding theory that are needed in Sections 3, 4, and 5. A thorough introduction to the subject can be found in, e.g., MacWilliams and Sloan [39] or McEliece [40].

A *code* over the alphabet Σ is a function E from Σ^k to Σ^n. A *binary code* is one in which $\Sigma = \{0, 1\}$. Words in the domain of E are *messages*, and words in the image of E are *codewords*. If B and B' are two elements of Σ^n, the *distance* between them is the number of places in which they differ. Computing $E(A)$, given the message A, is called *encoding*. Given a word $B \in \Sigma^n$, determining whether there exists a message A such that $E(A) = B$ is called *error detection*. Finding a valid codeword B' that is closest to B is called *error correction*. If B is a codeword, then finding the message A such that $E(A) = B$ is called *decoding*.

The *minimum distance* of a code E is the minimum, over all pairs of codewords, of the distance between them. The *relative distance* between codewords B and B' is the distance between B and B' divided by n, and the relative distance of the code E is its minimum distance divided by n. The ratio k/n is called the *rate* of the code.

An (n, k) *linear code* C over the finite field F is a k-dimensional subspace of the n-dimensional vector space over F. A *generator matrix* G for C is a $k \times n$ matrix whose k rows are linearly independent vectors in F^n. To encode a word $x \in F^k$, simply perform the matrix multiplication $x^T G$. If $F = \mathrm{GF}(2)$, then C is called a *binary linear code*.

The following lemma states a fundamental fact about polynomials that is important in our uses of coding theory.

Schwartz's Lemma [51]: Two distinct m-variable polynomials over the finite field F, each of total degree at most d, must assume different values on at least $(1 - d/|F|) \cdot |F|^m$ of the $|F|^m$ points in their domain.

An equivalent statement of Schwartz's Lemma is that a polynomial in $F[X_1, \ldots, X_m]$ of total degree at most d is either identically zero or has at most $d \cdot |F|^{m-1}$ distinct roots. This fact is well known in the case $m = 1$; the proof of the more general fact is by induction on m.

In a *polynomial code* over a finite field F, the messages $A = (a_1, \ldots, a_k)$ are interpreted as coefficients of a polynomial $p(x_1, \ldots, x_m)$ over F. The codeword corresponding to A is obtained by writing down the value of p at each m-tuple in F^m or in some subset of F^m. A fundamental example of polynomial codes are those of Reed and Solomon. A message $A = (a_0, \ldots, a_k)$, where $a_i \in F$, is interpreted as the univariate polynomial $p(x) = a_0 + a_1 x + \cdots + a_k x^k$ in $F[x]$, where $|F| = n = ck$, for some constant $c > 1$. It is encoded by writing down the value of p at each of the points $b_1, \ldots, b_n$ in F.

Because two degree-k polynomials can agree on at most k points, the minimum distance of this code is at least $(c - 1)k = \Omega(n)$. Note, however, that this is "distance over F" in that two symbols in a codeword are considered "different" if they differ as field elements. If we view this as a binary code, then the distance between two codewords is the number of bits in which they differ; because $\log n$ bits are required to write down a field element, the relative distance of the code can tend to 0 as n grows.

Justesen [32] uses Reed-Solomon codes to construct an infinite family of binary, linear codes of constant rate and constant relative distance. Justesen's basic trick is to compose the Reed-Solomon code with codes in which the messages have length $\log n$, and both the rate and the relative distance are constant. It is known that, with high probability, if b is chosen uniformly at random from F, the mapping $x \mapsto (x, bx)$ has constant relative distance. (All such maps have rate 1/2 and messages of length $\log n$.) Let $E_i(x) = (x, b_i x)$. Then the Justesen encoding E is defined as

$$E(a_0, \ldots, a_k) \equiv (E_1(p(b_1)), \ldots, E_n(p(b_n))),$$

where $p(x) = a_0 + a_1 x + \cdots + a_k x^k$. Because a high fraction of the codes E_i have constant relative distance, E has constant relative distance, for $n \to \infty$. For more details about Reed-Solomon and Justesen codes, see [32, 39, 40].

Finally, we recall the Berlekamp-Welch decoding theorem.

Berlekamp-Welch Decoding [11]: Given pairs (x_1, y_1), ..., (x_n, y_n) of points in a finite field F, there is an algorithm that finds a univariate polynomial p of degree at most d such that $p(x_i) = y_i$ for all but k pairs (x_i, y_i), provided $2k + d < n$ and such a p exists. The running time of the algorithm is polynomial in n and l, where l is the size of elements of F.

The Berlekamp-Welch algorithm and a complete proof of its correctness are given in the paper of Gemmell and Sudan [28]. The essence of the algorithm is to find polynomials w and p such that $degree(w) \leq k$, $degree(p) \leq d$, w is not identically zero, and $w(x_i) \cdot y_i = w(x_i) \cdot p(x_i)$, for $1 \leq i \leq n$. Berlekamp and Welch reduce this task to that of finding polynomials w and q such that $degree(w) \leq k$, $degree(q) \leq k + d$, w is not identically zero, $w(x_i) \cdot y_i = q(x_i)$ for $1 \leq i \leq n$, and w divides q. They show that two pairs of polynomials q, w and l, u that satisfy the first four requirements also satisfy $q/w = l/u$. Thus one can translate the first four requirements plus the requirement that w divide q into a set of linear equations in the coefficients of q and w and find an arbitrary solution to this set of equations.

3 Using Good Linear Codes to Improve Computational Efficiency

This section provides examples of how coding theory is used to improve computational efficiency. What is needed is an infinite family $\{E_{k_i}\}_{i \geq 1}$ of binary linear codes that have constant rate, constant relative distance, and a uniform efficient encoding algorithm. That is, there is one FP encoding function that, for any k_i, takes a string x in $\{0,1\}^{k_i}$ and produces $E_{k_i}(x)$. Note that efficient decoding, although not a disadvantage, is not needed in this application. Linear code families with these properties exist, and in fact the Justesen codes described in Section 2.2 above are such a family.

3.1 Efficient Use of Randomness as a Resource

Randomized algorithms play an important role in both theory and practice of computation. As first suggested by Karp and Pippenger [35], it is complexity theoretically natural to view random bits as a resource analogous to time or space and to design algorithms that use as few of them as possible. There is also a practical motivation to use few random bits, because such bits have to be generated by a physical device, such as a Geiger counter or a Zener diode, and these devices are slow. We now describe an elegant construction of Naor and Naor [42] that uses good linear codes to save random bits.

Suppose that a probabilistic polynomial-time machine M uses n independent random bits. This means that it samples from a probability space of size 2^n. Naor and Naor use linear codes to construct a smaller probability space that, for certain machines M, can be used to solve the same problem that M solves. Because this space is smaller, the machine that samples from it uses fewer independent random bits than M uses. The starting point for their construction is the equivalence of the following two statements.

1. The $\{0,1\}$-random variables $x_1, \ldots, x_n$ are independent and, for all i, $\text{Prob}[x_i = 0] = \text{Prob}[x_i = 1] = 1/2$.

2. For every nonempty subset S of $\{1, \ldots, n\}$, the probability that $\sum_{i \in S} x_i = 1$ and the probability that $\sum_{i \in S} x_i = 0$ are both $1/2$. Here Σ is the mod-2 sum.

Suppose that statement 2 is relaxed to require only that every nonempty subset S of $\{1, \ldots, n\}$ be *ϵ-biased*. This means that

$$\left| \text{Prob}[\sum_{i \in S} x_i = 0] - \text{Prob}[\sum_{i \in S} x_i = 1] \right| \leq \epsilon.$$

Then $x_1, \ldots, x_n$ are called ϵ-biased random variables, and the sample space that they give rise to is of size $2^{O(\log(n)+\log(1/\epsilon))}$. If ϵ is $1/poly(n)$, then the sample space associated with these variables is polynomially small.

Let v and r be elements of $\{0,1\}^n$, and let $v \cdot r$ be the usual mod-2 dot product. The vector r is said to be a *distinguisher with respect to* v if $v \cdot r = 1$. Naor and Naor's construction of ϵ-biased random variables has three stages. In the first stage, they construct a set $\mathcal{F}$ of vectors in $\{0,1\}^n$ that can be sampled once using $O(\log(n))$ random bits; this set has the additional property that, for any nonzero $v \in \{0,1\}^n$, an r chosen uniformly at random from $\mathcal{F}$ is a distinguisher with respect to v with probability at least β, for some constant $\beta > 0$. In the second stage, these variables are sampled l times, where l depends on ϵ. In the third stage, these l samples are combined in a way that produces an ϵ-biased set of variables. The second and third stages do not use coding theory and hence are not explained here; refer to [42] for a complete explanation.

The theory of combinatorial and spherical designs provides another way to view the construction of $\mathcal{F}$. Geometrically, the set $v^\perp \equiv \{r \mid r \cdot v = 0\}$ is a hyperplane in the vector space $\{0,1\}^n$. A set of random vectors would be split equally between $v^\perp$ and its complement. The set $\mathcal{F}$ approximates the behavior of a random set. Approximations of random sets motivate many designs in this field; see, for example, the article in this volume by Sloane [55].

Let E_n be a binary linear code with constant rate and constant relative distance. That is, $E_n : \{0,1\}^n \longrightarrow \{0,1\}^m$, where $m = O(n)$, and the minimum weight of a codeword in $\{0,1\}^m$ is βm, for some constant $\beta > 0$. Let G_n be the generator matrix for E_n, and let $\mathcal{F}$ be the set of columns of G_n. The fact that, for any nonzero $v \in \{0,1\}^n$, the codeword $E_n(v)$ has weight at least βm means precisely that, for any such v, a column r of G_n chosen uniformly at random is a distinguisher with respect to v with probability at least β. Because $m = O(n)$, a column of G_n can be chosen uniformly at random using $O(\log(n))$ random bits. (Actually, for this property, it would suffice to have $m = poly(n)$.)

This construction can be generalized to allow v and r to be vectors over an arbitrary ring, rather than over GF(2), and $v \cdot r$ to denote the dot product in this ring. This generalization is applied in many areas of computer science, including combinatorial algorithms, parallel computation, fault diagnosis, and communication complexity. (See [42, Section 3].) Here we discuss two applications that entail saving random bits in a probabilistic algorithm.

Suppose that three $n \times n$ boolean matrices A, B, and C are given, and we wish to determine whether $AB = C$ without performing matrix multiplication. (The fastest known matrix multiplication algorithms have asymptotic complexity $O(n^c)$, where $2 < c < 3$, but the constants implied in the $O()$ notation are so huge that these algorithms run more slowly than straightforward $O(n^3)$ algorithms for any plausible size n; thus the "real" time complexity of current matrix multiplication methods is $O(n^3)$.) Freivalds [26] suggests the following probabilistic test: Choose a vector r uniformly at random from $\{0,1\}^n$ and check whether $(r^T A)B - r^T C = 0$. This test has time complexity $O(n^2)$, uses n random bits, always says "yes" if $AB = C$, and says "no" with probability at least $1/2$ if $AB \neq C$. The proof that this test says "no" with probability at least $1/2$ when $AB \neq C$ amounts to the observation that, if v is a nonzero column vector in $AB - C$, then an r chosen uniformly at random from $\{0,1\}^n$ is a distinguisher for v with probability at least $1/2$. Thus, to reduce the number of random bits needed by Freivalds's test to $O(\log(n))$, while maintaining the $O(n^2)$ running time and the $1/2$ probability of detecting an error, we need a set of vectors in $\{0,1\}^n$ that can be sampled using $O(\log(n))$ bits with the property that, for an arbitrary v, the sampled vector r is a distinguisher for v with some constant probability $\beta > 0$. The set $\mathcal{F}$ constructed above with linear codes has exactly this property. Explicitly, on $n \times n$ matrices, the improved algorithm chooses a column r of G_n uniformly at random and uses it exactly as the vector r is used in Freivalds's original algorithm. Because G_n is an $n \times m$ matrix and $m = O(n)$, the vector r is of length n, and the sampling requires $\log m = O(\log n)$ random bits.

The second example also concerns verification that a computation has been done correctly, this time in a finite field. The input is a prime p, a number a, and a list of pairs (x_1, y_1), $\ldots$, (x_n, y_n). We wish to verify that $a^{x_i} = y_i \bmod p$, for $1 \leq i \leq n$. The following test, due to Fiat and Naor, is presented in [42]: Choose $r = (r_1, \ldots, r_n)$ uniformly at random from $\{0,1\}^n$; let $t = \sum_{i=1}^n r_i x_i \bmod (p-1)$ and $m = \prod_{i=1}^n y_i^{r_i} \bmod p$; check whether $a^t = m \bmod p$. This test requires $O(n + \log(p))$ modular multiplications instead of the $O(n \log(p))$ required to check each equality separately. If all equalities hold, the test always says "yes," and if at least one does not hold, it says "no" with probability at least $1/2$. Its cost in random bits is

n. The proof that it detects a faulty input with probability at least $1/2$ reduces to a proof that a random r is a distinguisher for $w = (w_1, \ldots, w_n)$ with probability at least $1/2$, where $a^{z_i} = y_i \bmod p$ and $w_i = x_i - z_i \bmod (p-1)$. As in the previous example, sampling from the family $\mathcal{F}$ derived from a good linear code produces a distinguisher with constant probability and requires only $O(\log(n))$ random bits.

3.2 Efficient Use of Communication Bits

Cryptographic complexity theory studies the efficiency of procedures that protect privacy and integrity of information. Most of these tasks are accomplished by protocols that involve two or more communicating machines. To be of practical use, a protocol must be efficient not only in its use of time and space but also in its use of communication bandwidth. In this section, we show how good linear codes are used to lower the communication complexity of a fundamental cryptographic protocol.

A *bit-commitment* protocol, executed by two probabilistic polynomial-time machines called the *committer* C and the *receiver* R, has two stages:

- The *commit stage*: C has a bit b that she commits to R. C and R exchange messages, and after this exchange, R has some information that represents b.
- The *reveal stage*: C and R exchange messages and, after this exchange, R knows b.

The expense of the protocol, in terms of local computation costs of C and R and number of bits exchanged, depends on a "security parameter" n. That is, the guarantees embodied in the following two properties that the protocol must satisfy can be made stronger or weaker by choosing a larger or smaller n. Thus the input to the protocol is the pair $(b, 1^n)$, and it makes sense to say that C and R are probabilistic polynomial-time TMs. This statement would not make sense if the input were just the single bit b. The notation 1^n is used to mean a string of n 1's; it provides a way of writing the security parameter "in unary" so that the length of the input is n, and one can use "polynomial time" to mean "polynomial in n" as usual.

Let C' and R' denote probabilistic polynomial-time TMs that play the roles of C and R; they may follow the protocol faithfully, or they may try to cheat in order to gain some advantage over the other party. The protocol must satisfy the following two properties for all probabilistic polynomial-time C' and R', all polynomials p, and sufficiently large n.

- The *privacy property*: After the commit stage, b is private. That is, R' can guess b with probability at most $1/2 + 1/p(n)$.
- The *integrity property*: After the commit stage, b is fixed. That is, C' can reveal only the b that was committed to during the commit stage; if she tries to reveal the opposite bit, she is caught with probability at least $1 - 1/p(n)$.

Executing the commit stage is thus analogous to having C write b on a piece of paper, put the paper into a locked box to which only C has a key, and give the box to R. Executing the reveal stage is analogous to having C unlock the box. C knows that R cannot guess b, because only C has a key to the box. R knows that C has not changed the value of b after the commit stage, because the box has been in R's possession. Bit-commitment protocols are essential building blocks in user-authentication schemes and many of the other tasks that are needed in secure systems.

Let $m(n) > n$ be a polynomially bounded function. The probabilistic polynomial-time machine G is a *cryptographically strong pseudorandom number generator* if, for all polynomials p and all probabilistic polynomial-time machines A,

$$|\mathrm{Prob}[A(y) = 1] - \mathrm{Prob}[A(G(s)) = 1]| < \frac{1}{p(n)},$$

for all sufficiently large n, where y and s are chosen uniformly from $\{0,1\}^{m(n)}$ and $\{0,1\}^n$, respectively. Here A is a polynomial-time statistical test that is trying to distinguish the length-$m(n)$, pseudorandom output of G on a random seed s of length n from the truly random string y of length $m(n)$. The generator G is cryptographically strong if no such test A succeeds with nonnegligible probability.

Naor [41] shows how to use any such generator G to build a bit-commitment scheme. The importance of this result is that it provides flexibility in the design of secure systems that require bit-commitment, because pseudorandom generators have been proven to exist under very general conditions [29, 30]. In the following statement of Naor's protocol, $B_i(s)$ denotes the i^{th} bit, $1 \le i \le m(n)$, in the pseudorandom sequence $G(s)$ generated from random seed $s \in \{0,1\}^n$. The symbol $\oplus$ denotes the exclusive-or operation; it can be applied to a single bit or, componentwise, to a vector of bits. In the following protocol, we can set $m(n) = 3n$, because this is sufficient to make the chance of catching a cheating C' at least $1 - 2^{-n}$. We will modify this choice of m below when we design a protocol to commit to many bits simultaneously.

Bit-Commitment$(b, 1^n)$:

- Commit stage:
  1. R chooses $r = (r_1, \ldots, r_{3n})$ uniformly at random from $\{0,1\}^{3n}$ and sends r to C.
  2. C chooses s uniformly at random from $\{0,1\}^n$ and sends $d = (d_1, \ldots, d_{3n})$ to R, where $d_i = B_i(s)$ if $r_i = 0$ and $d_i = B_i(s) \oplus b$ if $r_i = 1$.
- Reveal stage: C sends b and s to R, who verifies that all of the bits of d are correct.

The proof that this protocol satisfies the definition of bit-commitment is give in [41].

Linear codes come into the picture when C needs to commit to many bits at once. In fact, this is exactly what is needed in practical applications of bit commitment. When committing to one bit using the above protocol, C and R incur a communication cost of $O(n)$, where n is the security parameter. A naïve use of the protocol to commit to a set of bits $b_1, \ldots, b_k$ would incur a communication cost of $O(kn)$. Codes allow us to reduce this cost substantially.

For simplicity of exposition, suppose that $k = 3n/2$; the following construction actually works whenever $k = O(n)$. As in the previous section, $E_k : \{0,1\}^k \longrightarrow \{0,1\}^{\alpha k}$ is a linear code with constant rate $1/\alpha$ and constant relative distance β. More precisely, there is an infinite family of such codes, one for each k of the form $3n/2$, $n \geq 1$, and an FP function that, on input $x \in \{0,1\}^k$, computes $E_k(x)$. We require that $\alpha k \log(2/(2-\beta))$ be at least $3n$, and once again the Justesen codes satisfy these requirements.

Let $G : \{0,1\}^n \longrightarrow \{0,1\}^{m(n)}$ be a cryptographically strong pseudorandom generator for some $m(n) \geq 3\alpha n$; in the following discussion, we will only be concerned with the first $3\alpha n = 2\alpha k$ bits of a pseudorandom sequence $G(s)$, $s \in \{0,1\}^n$. For any t, $\alpha k \leq t \leq m(n)$ and any 0-1 vector $r = (r_1, \ldots, r_t)$ in which exactly αk of the r_i's are 1, let $G_r(s)$ denote the vector $(a_1, \ldots, a_{\alpha k})$, where $s \in \{0,1\}^n$ is a random seed, $j(i)$ is the index of the i^{th} 1 in r, and $a_i = B_{j(i)}(s)$.

Many-Bit-Commitment$((b_1, \ldots, b_k), 1^n)$:

- Commit stage:
  1. R chooses $r = (r_1, \ldots, r_{2\alpha k})$ uniformly at random from the set of vectors in $\{0,1\}^{2\alpha k}$ in which exactly αk of the r_i's are 1. R sends r to C.
  2. C computes $c = E_k(b_1, \ldots, b_k)$. C chooses s uniformly at random from $\{0,1\}^n$ and computes $e = c \oplus G_r(s)$. C sends to R the vector e and the bit $B_i(s)$ for every i, $1 \leq i \leq 2\alpha k$, such that $r_i = 0$.
- Reveal stage: C sends $(b_1, \ldots, b_k)$ and s to R, who verifies that all of the bits he received are correct.

The number of bits exchanged during the execution of Many-Bit-Commitment is $O(\max(k, n))$, which is as promised a substantial improvement over $O(kn)$. The proof that it has the privacy property uses the definition of pseudorandom number generation and does not involve codes. The proof that it satisfies the integrity property starts with the observation that C' can only cheat if she can find two seeds s and s' in $\{0,1\}^n$ and an input sequence $(b'_1, \ldots, b'_k) \neq (b_1, \ldots, b_k)$ such that $B_i(s) = B_i(s')$ whenever $r_i = 0$ and the sequences $G_r(s) \oplus E_k(b_1, \ldots, b_k)$ and $G_r(s') \oplus E_k(b'_1, \ldots, b'_k)$ are identical. Consider any pair of seeds s, s'. Because the distance between the codewords $E_k(b_1, \ldots, b_k)$ and $E_k(b'_1, \ldots, b'_k)$ is at least $\alpha\beta k$, the pseudorandom sequences $G_r(s)$ and $G_r(s')$ must also be at least $\alpha\beta k$ apart for C' to be able to cheat successfully. This means that there are at least $\alpha\beta k$ indices i for which $B_i(s) \neq B_i(s')$. The indices i for which $r_i = 0$ form a random subset of $\{1, \ldots, 2\alpha k\}$ of size αk; thus, the probability that all such i satisfy $B_i(s) = B_i(s')$ is at most

$$\left(\frac{2\alpha k - \alpha\beta k}{2\alpha k}\right)^{\alpha k} = \left(1 - \frac{\beta}{2}\right)^{\alpha k}.$$

Because $\alpha k \log(2/(2-\beta)) \geq 3n$, this probability is at most 2^{-3n}. Now multiply by 2^{2n}, the total number of pairs s, s', to see that the probability that C' can convince R to accept a wrong sequence is at most 2^{-n}.

Good linear codes are used in a very similar way in the *digital signature* schemes of Even, Goldreich, and Micali [23] and Dwork and Naor [22]. Digital signature schemes consist of a *signing* algorithm S and a *verification* algorithm V. They capture the paper world's notion of "signature" in that, if signer A constructs a signature $s = S(x)$ of document x, then, by running V, anyone can verify that s is a legitimate signature of x and that it was A who constructed it. Because signatures are often verified by resource-limited devices like smart cards, it is important that signature schemes make efficient use of random bits and communication bits.

4 Program Testing and Correction

This section presents a natural complexity theoretic analogue of error-detection and -correction, namely the testing and correction of computer programs. Program testers and correctors are similar to error-detecting and -correcting codes in their overall purpose: A tester is supposed to output PASS or FAIL, depending upon whether the program is correct; a corrector is supposed to take an input x and a program P that has been declared by the tester to be at least "nearly correct" and output the correct value for $P(x)$. Before giving a formal definition of program testers and correctors, we note two ways in which they are different from traditional error-detecting and -correcting codes.

The first difference is that only asymptotically good testers and correctors matter. A "program" is assumed to compute a function whose domain is infinite. When we speak of the "running time" of the program, it is implicit that we mean the asymptotic running time. As we will see in the formal definition given below, a tester or corrector is supposed to work on the entire domain of the function. Thus, while specially designed codes $E : \Sigma^k \longrightarrow \Sigma^n$ for particular small values of k and n are of interest in coding theory, even when they are not part of an infinite family of codes with the same properties, testers and correctors that only work on some finite subdomain of the program in question are not even defined.

The second difference is that probabilistic algorithms play a much more central role in program testing and correction than they do in traditional coding theory. A program can pass the tester if it is correct on most of its inputs, but not all, and a corrector guarantees only that the program is correct with high probability on each input, not that it is correct with probability one. Tolerance for a small probability of error allows for the development of very efficient testing and correction algorithms and the application of these algorithms in seemingly unrelated problem areas in complexity theory, as explained in Section 5.2 below.

The precise definition of program testing and correction that we will use here was given by Babai, Fortnow, and Lund [6], who built upon the general approach first taken by Blum, Luby, and Rubinfeld [14]. The work in [14] was in turn inspired by work on program "checking" by Blum and Kannan [12, 13, 33]. A thorough treatment of all of these notions can be found in, e.g., [1, 25, 33, 47, 57].

The probabilistic polynomial-time oracle Turing machines T and C form a *self-testing/correcting pair for f* if they behave as follows. The output of T is always PASS or FAIL. For any n, $T^f(1^n)$ = PASS with probability at least 3/4. If the probability that $T^Q(1^n) = \text{PASS}$ is at least 1/4, then for any input x of size n, the probability that $C^Q(x) = f(x)$ is at least 3/4. In other words, the tester T takes a program Q that purports to compute f and a size parameter n, written in unary. If Q is an everywhere-correct program for f, then T should output PASS with high probability, for all sizes n. On the other hand, if there is a nonnegligible probability that T passes Q on input size n, then Q may in fact have errors somewhere, but for any input x of size n, the corrector C must be able, possibly by making repeated calls to the program Q, to produce the right value $f(x)$ with high probability. As usual, the 3/4 probability of correctness and the 1/4 threshhold for correctability can be increased or decreased if one is willing to use slower, but still polynomial-time, testers and correctors.

Rubinfeld and Sudan [49] generalize the notion of a tester for a function f to that of a tester for a function family $\mathcal{F}$. Let f and g be two functions defined on the same domain, and let $d_n(f,g)$ denote the fraction of inputs of size n on which functions f and g differ. If $\mathcal{F}$ is a family of functions defined on the same domain as the function f, then $\Delta_n(f,\mathcal{F})$ is the minimum, over all $g \in \mathcal{F}$, of $d_n(f,g)$. The probabilistic polynomial-time oracle Turing machine T is a *tester for the function family* $\mathcal{F}$ if, for all n and all $f \in \mathcal{F}$, $T^f(1^n)$ outputs PASS with probability at least 3/4 and, for all n and all programs Q such that $\Delta_n(Q,\mathcal{F}) \geq 1/4$, $T^Q(1^n)$ outputs FAIL with probability at least 3/4.

Blum, Luby, and Rubinfeld [14] provided the first example of a family of functions that can be tested and corrected – linear functions over finite fields. We describe this example here, because it is used in Section 5.1 below. Recall that f is a linear, n-variable function over the finite field F if there are n coefficients $a_1, \ldots, a_n$ in F such that

$$f(x_1,\ldots,x_n) = \sum_{i=1}^{n} a_i \cdot x_i,$$

for $(x_1,\ldots,x_n) \in F^n$. Let $\{F_n\}_{n\geq 1}$ be a sequence of finite fields, $\mathcal{F}_n$ be the family of linear, n-variable functions over F_n, and $\mathcal{F} = \cup_{n\geq 1}\mathcal{F}_n$. Blum, Luby, and Rubinfield observe the following useful fact.

Linearity testing and correcting [14]: Suppose that $0 \leq \delta \leq 1/3$ and that the function $g : F_n^n \longrightarrow F_n$ has the property that two elements x and y both chosen uniformly at random from F_n^n satisfy the equation $g(x+y) = g(x) + g(y)$ with probability at least $1 - \delta/2$. Then $\Delta_n(g,\mathcal{F}) \leq \delta$. On the other hand, if $\Delta_n(g,\mathcal{F}) \leq \delta$, then, for any $x \in F_n^n$, the probability is at least $1 - 2\delta$ that a y chosen uniformly at random from F_n^n satisfies $g(x) = g(x+y) - g(y)$.

From this basic observation, it is straightforward to construct a self-testing/correcting pair for any sequence of linear functions over finite fields. The essence of the testing strategy is to check that the program Q satisfies the identity $Q(x+y) = Q(x) + Q(y)$ at a constant number of random sample points and that it satisfies some "initial conditions" that define the particular linear function that Q is supposed to compute. Exactly what this constant number of sample points is depends on the error probability one is willing to tolerate. The fact that these testers and correctors require only a constant number of calls to the program being tested or corrected is crucial in Section 5 below. We give some details about a more general tester, i.e., one of the total-degree, multivariate polynomial testers provided by Rubinfeld and Sudan [49]. More details about the linearity tester can be found in [14].

These testers for multivariate linear functions and polynomials are based on "robust characterizations" of function families. The basic idea of such a characterization is to take an exact characterization of the family (such as $g(x+y) = g(x) + g(y)$ for the family of linear functions) and prove that the "for all" quantifier of the variables x and y can be replaced by a "for most" quantifier. A robust characterization can be used to build a tester, because polynomially many samples suffice to show that a property holds for most elements of the domain of the function, whereas the entire (exponential-sized) domain would have to be tested to establish that the property held for all elements. Rubinfeld and Sudan [49] provide a detailed treatment of this notion as well as several alternative robust characterizations of multivariate polynomials that can be used to build testers. Subsequent robust characterizations of some non-algebraic function classes, as well as program testers and correctors based on these characterizations, can be found in [48].

Let $\{\mathrm{GF}(p_n)\}_{n\geq 1}$, where p_n is prime, be a sequence of finite fields. Let $\mathcal{F}_n$ be the family of n-variable polynomials over $\mathrm{GF}(p_n)$ that have total degree at most d_n, and let $\mathcal{F}$ denote $\cup_{n\geq 1}\mathcal{F}_n$. We require that p_n be at least $d_n + 2$. Throughout this discussion, the coefficient $\alpha_{n,i} = (-1)^{i+1}\binom{d_n+1}{i}$. The computation of $\alpha_{n,i}$, as well as all other computations in this example, is done modulo p_n. Note that $\alpha_{n,i}$ depends only on i and the degree d_n, not on the particular polynomial being tested.

The starting point for this tester is the following (well known) exact characterization of multivariate polynomials. The fundamental algebra underlying this and other exact characterizations of polynomials can be found in, e.g., the classic book of Van der Waerden [21].

Evenly spaced points characterization – exact: The function $g_n : \mathrm{GF}(p_n)^n \longrightarrow \mathrm{GF}(p_n)$ is in $\mathcal{F}_n$ if and only if $g_n(x) = \sum_{i=1}^{d_n+1} \alpha_{n,i} g_n(x+iy)$, for all x and y in $\mathrm{GF}(p_n)^n$.

The crucial fact used by the tester is that this characterization is robust.

Evenly spaced points characterization – robust [49]: Let $\delta_0 = 1/2(d_n+2)^2$. If the function $Q_n : \mathrm{GF}(p_n)^n \longrightarrow \mathrm{GF}(p_n)$ has the property that

$$\delta \equiv \mathrm{Prob}(Q_n(x) \neq \sum_{i=1}^{d_n+1} \alpha_{n,i} Q_n(x + i\cdot y)) \leq \delta_0,$$

where the probability is induced by (independently) choosing x and y uniformly at random from $\mathrm{GF}(p_n)^n$, then $\Delta_n(Q_n, \mathcal{F}) \leq 2\delta$.

Let $g_n(x) \equiv \mathrm{maj}_{y\in \mathrm{GF}(p_n)^n}\{\sum_{i=1}^{d_n+1} \alpha_{n,i} Q_n(x+iy)\}$, where "maj" of a multiset picks the most common element, breaking ties arbitrarily. The fact that g_n is within distance 2δ of Q_n follows immediately from the definitions of g_n and δ. Rubinfeld and Sudan [49, Section 4] provide a clever proof that $g_n \in \mathcal{F}_n$. First they show that, for all x,

$$\mathrm{Prob}(g_n(x) = \sum_{i=1}^{d_n+1} \alpha_{n,i} Q_n(x+iy)) \leq 1 - 2(d_n+1)\delta, \tag{1}$$

where the probability is induced by choosing y uniformly at random from $\mathrm{GF}(p_n)^n$. This implies that

$$\mathrm{Prob}(g_n(x+iy) = \sum_{j=1}^{d_n+1} \alpha_{n,j} Q_n((x+iy)+j(h_1+ih_2))) \geq 1 - 2(d_n+1)\delta, \tag{2}$$

for any fixed i, where the probability is induced by (independently) choosing h_1 and h_2 uniformly at random from $\mathrm{GF}(p_n)^n$ – just set $y = h_1 + ih_2$ in Inequality (1). Furthermore for all $1 \leq j \leq d_n + 1$,

$$\mathrm{Prob}(\sum_{i=0}^{d_n+1} \alpha_{n,i} Q_n((x+jh_1)+i(y+jh_2)) = 0) \geq 1 - \delta, \tag{3}$$

where the probability is again computed over independently chosen h_1 and h_2. This follows from the definition of δ and the fact that $x + jh_1$ and $y + jh_2$ are both uniformly distributed over $\mathrm{GF}(p_n)^n$. We can use Inequalities (2) and (3) to show that g_n satisfies $\sum_{i=0}^{d_n+1} \alpha_{n,i} g_n(x+iy) = 0$, for all x and y. Because of the (exact) evenly spaced points characterization, this means that $g_n \in \mathcal{F}_n$.

In the following calculations, all probabilities are induced by choosing h_1 and h_2 independently and uniformly at random from $\mathrm{GF}(p_n)^n$. First replace $g_n(x+iy)$ by $\sum_{j=1}^{d_n+1} \alpha_{n,j} Q_n((x+iy)+j(h_1+ih_2))$. Then, by Inequality (2), we have

$$\mathrm{Prob}(\sum_{i=0}^{d_n+1} \alpha_{n,i} g_n(x+iy) \neq \sum_{i=0}^{d_n+1} \alpha_{n,i} \sum_{j=1}^{d_n+1} \alpha_{n,j} Q_n((x+iy)+j(h_1+ih_2))) \leq 2(d_n+1)(d_n+2)\delta. \tag{4}$$

Switch the order of summation and regroup the terms in the inner summation of Inequality (4). That give us

$$\mathrm{Prob}(\sum_{i=0}^{d_n+1} \alpha_{n,i} g_n(x+iy) \neq \sum_{j=1}^{d_n+1} \alpha_{n,j} \sum_{i=0}^{d_n+1} \alpha_{n,i} Q_n((x+jh_1)+i(y+jh_2))) \leq 2(d_n+1)(d_n+2)\delta. \tag{5}$$

For any fixed j, Inequality (3) gives us that the inside summation $\sum_{i=0}^{d_n+1} \alpha_{n,i} Q_n((x+jh_1)+i(y+jh_2))$ in Inequality (5) is 0 with probability at least $1-\delta$. Putting this all together, we see that the probability that $\sum_{i=0}^{d_n+1} \alpha_{n,i} g_n(x+iy)$ is 0 is at least $1-(2(d_n+1)(d_n+2)\delta+(d_n+2)\delta)$. Because $\delta \leq 1/2(d_n+2)^2$, this probability is positive. Note, however, that this probability is computed over random choices of h_1 and h_2, while the statement "the probability that $\sum_{i=0}^{d_n+1} \alpha_{n,i} g_n(x+iy)$ is 0" is independent of both h_1 and h_2. Thus, because this probability is positive, it must in fact be 1; the conclusion that $g_n \in \mathcal{F}_n$ follows.

From this robust characterization, it is straightforward to build a tester. The input to the tester consists of a size parameter 1^n, a program Q that purports to compute a sequence of multivariate polynomials $\{f_n\}_{n \geq q}$ in $\mathcal{F}$, a set of pairs $(x_1, f_n(x_1)), \ldots, (x_t, f_n(x_t))$ that define the particular polynomial that Q is supposed to compute on inputs of size n, and some constants that specify the error probability that we can tolerate in the tester. The tester first chooses $O(1)$ pairs (x, y) and checks for each one whether $\sum_{i=0}^{d_n+1} \alpha_{n,i} Q_n(x+iy) = 0$; if this check fails for a high enough fraction of the pairs (x, y), the tester outputs FAIL and quits. If it does not quit at this point, the tester has concluded that the function computed by Q on inputs of size n is sufficiently close to a polynomial in $\mathcal{F}_n$. It now proceeds to test whether Q is computing a function that is close to the particular polynomial defined by $(x_1, f_n(x_1)), \ldots, (x_t, f_n(x_t))$. For each j, $1 \le j \le t$, it (independently) chooses $O(1)$ points y uniformly at random from $\mathrm{GF}(p_n)^n$ and checks that $f_n(x_j) = \sum_{i=1}^{d_n+1} \alpha_i Q_n(x_j + iy)$. It PASSes Q if and only if a high enough fraction of these checks succeed. For details, including values of all of the relevant constants, see Rubinfeld and Sudan [49, Section 6.1].

The best-known multivariate correctors (which are indeed the best possible) are provided by Gemmell and Sudan [28]. They are based not on robust characterizations but rather on Berlekamp-Welch decoding.

Multivariate polynomial self-correction [28]: On inputs $x = (x_1, \ldots, x_n) \in F_n^n$, for some finite field F_n, a program Q purports to compute an n-variable polynomial g_n over F_n of degree d_n. Suppose that Q has passed a tester T on inputs of size n and that the particular guarantee given by T is that $\Delta_n(Q, g_n) \le 1/2 - \delta$. Suppose further that, for all n, $|F_n| = \Omega((1/\delta + d_n)^2)$. Then there is a self-corrector C for Q; that is, on input x, C outputs $g_n(x)$ with high probability.

Gemmell and Sudan's reduction of multivariate polynomial correction to Berlekamp-Welch decoding works as follows. They construct a subdomain J_n of F_n^n that is parameterized by a single variable t, i.e., $J_n = \{D(t) \mid t \in F_n\}$, that has the following properties:

1. The function $g_n(D(t))$ is a polynomial in t of degree $O(d_n)$.
2. The input point $x = (x_1, \ldots, x_n)$ is contained in J_n, and $D(0) = x$.
3. With high probability, the fraction of J_n on which Q differs from g_n is approximately $\Delta_n(Q, g_n)$. That is, sampling on J_n yields approximately the same error rate for Q as sampling on all of F_n^n.

Note that this construction suffices for the reduction, because the Berlekamp-Welch algorithm can be used to reconstruct the univariate polynomial $g_n(D(t))$, which can then be evaluated at $t = 0$.

The definition of J_n given by Gemmell and Sudan is beautifully simple: Let $D(t)$ be a random degree-2 curve that passes through x. That is, choose $2n$ elements α_1, β_1, $\ldots$, α_n, β_n each uniformly at random from F_n, and let the i^{th} coordinate of $D(t)$ be $\alpha_i t^2 + \beta_i t + x_i$. Because each coordinate of D is a polynomial of degree 2 in t, and $degree(g_n) = d_n$, it is clear that J_n satisfies property 1. Similarly, property 2 is satisfied trivially, because the i^{th} coordinate of $D(0)$ is $\alpha_i(0^2) + \beta_i(0) + x_i$, which is x_i. To establish property 3, Gemmell and Sudan show that J_n forms a pairwise independent sample of F_n^n and then apply Chebyshev's inequality; details can be found in [28].

This technique of replacing the n variables in a multivariate polynomial g_n by a random curve that passes through the point at which g_n is being evaluated has been used before in cryptographic complexity theory. For example, it enables computations on public servers in which the users do not have to reveal their private data – see, e.g., [7, 8, 10, 16]. The technique's intellectual roots are in the "secret sharing" scheme of Shamir [52]. In such a scheme, a "secret" must be distributed among n parties so that any $t+1$ of them can reconstruct it, but t or fewer cannot. This is accomplished in [52] by representing the secret s as an element in a finite field F, choosing t elements $\beta_1, \ldots, \beta_t$ uniformly at random from F, and giving each party a pair $(a, p(a))$, where $a \in F$ and $p(x) = \beta_t x^t + \cdots + \beta_1 x + s$; because $t+1$ points are required to determine the degree-t polynomial p (or even to infer any information about its constant term), the definition of a secret sharing scheme is satisfied.

We close this section by remarking that there is a self-testing/correcting pair for any function that is complete for FPSPACE or FEXP [6]. A detailed discussion of these structural complexity theoretic results on testing/correcting and of their relationship to the algebraic testers and correctors discussed in this section can be found in [25].

5 Using Polynomial Codes to Characterize Complexity Classes and to Prove Lower Bounds

Babai, Fortnow, and Lund [6] obtained a new characterization of the complexity class NEXP by showing that every language accepted by a nondeterministic exponential-time TM is also accepted by a multiprover interactive proof (MIP) system [9]. In an MIP system, two or more computationally unbounded "provers" convince a probabilistic polynomial-time "verifier" that an input string x is in a language L. If L is an arbitrary language in NEXP, then accepting computation paths of a nondeterministic TM for L have length exponential in the length of x, and it seems counterintuitive that the correctness of such a computation could be verified by a polynomial-time machine; in particular, such a machine cannot examine the entire computation. The provers in an MIP system overcome this by encoding an accepting computation using polynomial codes; the crucial property of a correct encoding is that it uses multivariate polynomials of the appropriate total degree over the appropriate field. The probabilistic polynomial-time verifier then uses multivariate polynomial testing techniques similar to those discussed in Section 4 in order to verify that the codewords correspond to an accepting computation. Such a tester proceeds by sampling the polynomial at random points in its domain and does not have to read the whole (exponential-length) encoding. In fact, the first multivariate polynomial tester was developed by Babai, Fortnow, and Lund [6] precisely for the purpose of encoding long accepting computations so that they could be verified by a probabilistic polynomial-time machine; the testers of Rubinfeld and Sudan [49] discussed in Section 4 came later and are more efficient than those in [6].

Prior to [6], the new characterization of PSPACE as the class of languages accepted by one-prover interactive proof systems [37, 53] had used simpler properties of polynomial codes. The results discussed in Section 5.1 refine the techniques of [6] considerably and represent the full flowering of the use of coding theory in new characterizations of traditional complexity classes.

From a coding theoretic point of view, it is striking that, while these results do need infinite families of polynomial codes that have constant relative distance, there is no need for the rate of the codes to be large. Indeed, the length of a codeword may be any polynomially bounded function of the length of a message.

In Section 5.2, we show how the characterizations of NP and PSPACE discussed in Section 5.1 can be used to prove lower bounds on the complexity of approximating certain combinatorial optimization functions. More precisely, they are used to show that there are ratios ϵ for which ϵ-approximating these functions is NP-hard or PSPACE-hard, i.e., just as hard as computing the function exactly. This is a "lower bound" in the same sense that an NP-completeness or PSPACE-completeness result is a lower bound – unless P = NP (resp. P = PSPACE), these approximation problems cannot be solved in polynomial time.

5.1 Probabilistically Checkable Proof Systems

A nondeterministic polynomial-time Turing machine M may be regarded as a "proof system" in which the polynomial-time verifier is deterministic, and the statements proved are of the form "$x \in L(M)$." If x is indeed in $L(M)$, then any accepting computation of M on input x is a "proof" of this fact, and if x is not in $L(M)$, then no such proof exists. The proofs provided by such a system are very fragile: If one bit in the description of a computation is changed, the verifier's decision to accept or reject x on this computation may also change. The goal of the proof systems defined in this section is to make proofs of membership in NP languages more robust: If a correct proof is altered slightly, it should still be recognizable as "essentially correct," and any input x that is not in the language should only give rise to computation paths that are "far from correct." This is closely related to the goals of error-detection and error-correction that are addressed by traditional coding theory.

Robust versions of NP proof systems were first sought explicitly by Babai et al. [5]. The formalism in [5] modifies the definition of a nondeterministic TM's proof system by allowing the verifier to be probabilistic and restricting its running time to polylogarithmic in n, the size of x. Because the running time of the verifier was sublinear in n, [5] explicitly required the input x to be presented via an error-correcting code. This was very influential and allowed [5] to achieve robust proofs as they are described above, but it is insufficient for the applications described in Section 5.2.

The membership proofs that we define here are called *probabilistically checkable proofs* and were first defined formally by Arora and Safra [4]. A language L is in PCP$(r(n), q(n))$ if there is a probabilistic polynomial-time machine V (called the verifier) with the following properties. The input to V is a pair (x, π); the string x is claimed to be a member of L, and π is the purported proof of this claim. During the course of its computation on input (x, π), V flips $O(r(n))$ coins and examines $O(q(n))$ bits of the proof string π, where n is the size of x. If $x \in L$, then there is a proof string π such that V outputs 1 with probability 1 on input (x, π). If $x \notin L$, then, for all strings π, V outputs 1 with probability at most 1/2 on input (x, π).

Note that, by definition, NP = PCP$(0, poly(n))$. That is, a proof system in which the verifier flips no coins and reads the entire proof string is simply an NP machine. (In this case, the probability is 0, not 1/2, that a string x not in L is accepted by the verifier.) The PCP

Theorem shows that there is a dramatic tradeoff between the parameters $r(n)$ and $q(n)$; by allowing the verifier to flip a small number of coins, one can drastically lower the number of "query bits" that the verifier requires.

PCP Theorem [3, 4]: $\mathrm{NP} = \mathrm{PCP}(\log(n), 1)$.

If a verifier is to detect that a prover's claim is invalid by examining only a constant number of bits, the proof must be encoded in a way that "spreads" errors throughout the proof string π. This is similar in spirit to the traditional design goals of error-detecting codes, and the testing algorithms presented in Section 4 go part of the way toward allowing us to achieve this goal for proofs of membership in NP languages. The full proof of the PCP Theorem also uses the following Composition Lemma; in fact, a few restrictions on probabilistically checkable proof systems are required if they are to satisfy this lemma, but we omit discussion of them here – see, e.g., [1, 3, 4, 57] for a detailed treatment.

Composition Lemma [4]: If L is in both $\mathrm{PCP}(r_1(n), q_1(n))$ and $\mathrm{PCP}(r_2(n), q_2(n))$, then there are constants c_1 and c_2 such that L is in $\mathrm{PCP}(r(n), q(n))$, where $r(n) = r_1(n) + r_2(q_1(n)^{c_1})$ and $q(n) = q_2(q_1(n)^{c_2})$.

Arora et al. [3] combine the following "long, robust proof system" and "transparent proof system" to get the PCP Theorem.

Long, Robust Proofs [3]: $\mathrm{NP} \subseteq \mathrm{PCP}(poly(n), 1)$.

Transparent Proofs [5, 6]: There is a constant c such that $\mathrm{NP} \subseteq \mathrm{PCP}(\log(n), \log(n)^c)$.

We give an overview of the $\mathrm{PCP}(poly(n), 1)$ proof system for the NP-complete language 3SAT. Details can be found in [1, 3, 57].

Recall that an instance of 3SAT consists of a collection $\{v_1, \ldots, v_n\}$ of boolean variables and a collection $\{C_1, \ldots, C_m\}$ of clauses. Each C_j is a disjunction of three literals, where a literal is either a variable or its negation. A "satisfying assignment" $(a_1, \ldots, a_n)$, $a_i \in \{0, 1\}$, is one that makes all of the clauses true simultaneously when v_i is assigned the truth value a_i; here 1 and 0 denote True and False. The long, robust proof system uses the fact that one can choose a random degree-3 polynomial $\phi_r \in \mathrm{GF}(2)[X_1, \ldots, X_n]$ in such a way that, if $(a_1, \ldots, a_n)$ is a satisfying assignment, then $\phi_r(a_1, \ldots, a_n) = 0$ with probability 1 and, if $(a_1, \ldots, a_n)$ is not a satisfying assignment, then $\phi_r(a_1, \ldots, a_n) = 0$ with probability at most 1/2; here 1 and 0 denote the elements of GF(2). This is done by "arithmetizing" each C_j: A positive literal v_i is arithmetized as $(1 - X_i)$ and a negative literal $\overline{v_i}$ as X_i, and the three arithmetized literals in a clause are multiplied; for example, the clause $C_j = v_1 \vee v_2 \vee \overline{v_3}$ is arithmetized as the monomial $\tilde{C}_j = (1 - X_1)(1 - X_2)X_3$. Note that $\tilde{C}_j$ evaluates to 0 for any assignment that satisfies C_j. To choose a random polynomial ϕ_r with the desired properties, choose r uniformly at random from $\{0, 1\}^m$ and let $\phi_r(X_1, \ldots, X_n) = \sum_{j=1}^{m} r_j \tilde{C}_j$.

Any vector $(a_1, \ldots, a_n)$ in $\mathrm{GF}(2)^n$ defines three linear functions $A : \mathrm{GF}(2)^n \longrightarrow \mathrm{GF}(2)$, $B : \mathrm{GF}(2)^{n^2} \longrightarrow \mathrm{GF}(2)$, and $C : \mathrm{GF}(2)^{n^3} \longrightarrow \mathrm{GF}(2)$ as follows.

$$A(x) \equiv \sum_{i=1}^{n} a_i x_i$$

$$B(y) \equiv \sum_{i=1}^{n}\sum_{j=1}^{n} a_i a_j y_{i,j}$$

$$C(z) \equiv \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n} a_i a_j a_k z_{i,j,k}$$

Explicit function tables for A, B, and C have size singly exponential in n. Thus the index of a table entry can be written down using $poly(n)$ bits.

The prover in a long, robust proof system is supposed to write down the tables corresponding to an assignment $(a_1, \ldots, a_n)$ that he claims satisfies the original 3SAT instance. Given three tables, the verifier V can test that they indeed represent three linear functions, or at least three functions that are close to linear. To do this, V uses the linearity testing method of Blum, Luby, and Rubinfeld [14] discussed in Section 4 above: Instead of making calls to a "program" that purports to compute a linear function, V simply looks up the functional values in the table provided by the prover. If the tables pass this test, then V still cannot be sure that they represent linear functions completely accurately; if he could be sure of this, then he could obtain any desired functional value $A(x)$, $B(y)$, or $C(z)$ simply by looking it up. However, the definition of a tester does guarantee that, if the tables pass the test, there are unique linear functions A, B, and C that are very close to the functions given by the tables; furthermore, V can obtain accurate functional values $A(x)$, $B(y)$, or $C(z)$ by using the linearity self-corrector of [14]. It is crucial that the resources needed for linearity testing and correcting are exactly those available to the verifier in a PCP($poly(n), 1$) proof system: $poly(n)$ random bits are required to specify random elements x, y, and z in the domains of A, B, and C, but only $O(1)$ table entries need be queried, each of which is a single bit, because the range of A, B, and C is GF(2).

V can also use $poly(n)$ random bits and $O(1)$ query bits to verify that the linear functions A, B, and C determined by a legitimate set of tables are related to each other in the proper way. Interestingly, a convenient way to do this uses the Freivalds matrix product technique discussed in Section 3.1. Let $\hat{a} = (a_1, \ldots, a_n)$, $\hat{b} = (b_{1,1}, \ldots, b_{n,n})$, and $\hat{c} = (c_{1,1,1}, \ldots, c_{n,n,n})$ be the coefficient vectors for these functions. What must be verified is that $\hat{b} = \hat{a} \circ \hat{a}$ and $\hat{c} = \hat{a} \circ \hat{b}$, where $\circ$ denotes the outer product operation. Both $\hat{a} \circ \hat{a}$ and $\hat{b}$ can be viewed as $n \times n$ matrices; denote them by M_1 and M_2. Let r and s be uniformly chosen random vectors in $\mathrm{GF}(2)^n$. If $M_1 \neq M_2$, then $r^T M_1 s \neq r^T M_2 s$ with probabity at least 1/4. By definition, $r^T M_1 s = A(r) \cdot A(s)$, where $\cdot$ is just multiplication in GF(2) and $r^T M_2 s = B(r \circ s)$. Thus, if there is a flaw in the relationship between A and B, V can detect it with probability at least 1/4 by choosing two polynomial-length random vectors and evaluating three functional values; these evaluations can be done with the linearity corrector and require only a constant number of queries to the tables. The test can be repeated a constant number of times to increase the probability of detecting flaws. The test that $\hat{c} = \hat{a} \circ \hat{b}$ is analogous.

It remains to be seen how V can use the linear functions A, B, and C to test that the assignment $(a_1, \ldots, a_n)$ from which they were derived satisfies the original 3SAT instance. The essential point is that A, B, and C permit V to evaluate any degree-3 polynomial $f \in \mathrm{GF}(2)[X_1, \ldots, X_n]$ at the point $(a_1, \ldots, a_n)$. For any such f, there are index sets

$S_1 \subseteq \{1,\ldots,n\}$, $S_2 \subseteq \{(1,1),\ldots,(n,n)\}$, and $S_3 \subseteq \{(1,1,1),\ldots,(n,n,n)\}$ and a constant term $\alpha \in GF(2)$ such that

$$f(a_1,\ldots,a_n) = \alpha + \sum_{i\in S_1} a_i + \sum_{(i,j)\in S_2} a_i a_j + \sum_{(i,j,k)\in S_3} a_i a_j a_k.$$

Using the definition of A, B, and C, we can rewrite this as

$$f(a_1,\ldots,a_n) = \alpha + A(\chi(S_1)) + B(\chi(S_2)) + C(\chi(S_3)).$$

Note that α, S_1, S_2, and S_3 are determined completely by which of the coefficients of the polynomial f are 0 and which are 1; that is, they do not depend on the point $(a_1,\ldots,a_n)$ at which f is being evaluated.

To summarize, the PCP($poly(n)$, 1) proof system for 3SAT proceeds as follows. To prove that the instance $(\{v_1,\ldots,v_n\},\{C_1,\ldots C_m\})$ is satisfiable, the prover takes a satisfying assignment $(a_1,\ldots,a_n)$ and constructs the corresponding function tables A, B, and C. The verifier checks that the tables have all of the desired properties, using the linearity tester and corrector and the Freivalds technique as explained above. If any of these checks fails, V outputs 0 and stops. If A, B, and C have all of the desired properties, V chooses r uniformly at random from $\{0,1\}^m$ and derives the corresponding degree-3 polynomial ϕ_r from the instance $(\{v_1,\ldots,v_n\},\{C_1,\ldots C_m\})$. V then uses the tables A, B, and C to evaluate $\phi_r(a_1,\ldots,a_n)$ and outputs 1 if and only if $\phi_r(a_1,\ldots,a_n) = 0$.

The construction of PCP($\log(n), \log(n)^c$) proof systems for NP is quite intricate and is beyond the scope of this article. Multivariate polynomial testing plays a crucial role. The terminology of PCP had not yet been developed when [5, 6] were written, but a thorough explanation of the relationships among multivariate polynomial testing, transparent proofs, and PCP, along with a translation of the results of [5, 6] into PCP terms can be found in, e.g., [1, 4, 46, 49, 57]. As in the long, robust proof system, the proof strings π encode witnesses from the original NP machine for the language via polynomial codes. Here the codes have higher degree and a smaller number of variables than those in the long, robust proofs.

Two applications of the Composition Lemma now give the PCP Theorem. First let $r_1(n) = r_2(n) = \log(n)$ and $q_1(n) = q_2(n) = \log(n)^c$. That yields NP $\subseteq$ PCP($\log(n), \log\log(n)^{c'}$), for some constant c'. Then let $r_1(n) = \log(n)$, $q_1(n) = \log\log(n)^{c'}$, $r_2(n) = poly(n)$, and $q_2(n) = 1$; composing these two sets of parameters yields the theorem. (To be precise, it yields one direction of the theorem, namely NP $\subseteq$ PCP($\log(n)$, 1); the inclusion PCP($\log(n)$, 1) $\subseteq$ NP follows trivially from the definitions of the two classes.)

For some potential applications, it is desireable to have PCP($\log(n)$, 1) proof systems in which the proof strings π are as short as possible. For example, automated theorem-proving tools often produce proofs that are too long to be verified by a human reader. PCP techniques may provide an approach to this problem: Encode the automatically generated proof via the PCP Theorem, and have a PCP verifier check it by reading only a constant number of bits. Because theorem-proving tools produce proofs that are already too long to store and transmit conveniently, it is important that the PCP encoding process increase the proof length as little as possible. The construction of Polishchuk and Spielman [46] can be combined with the improved multivariate polynomial testers of [49] to construct PCP($\log(n), 1$) proof systems in which the proof strings are considerably shorter than those in [3].

Although less central than NP, the complexity class PSPACE also plays a prominent role in theoretical computer science, and many natural problems are PSPACE-complete. Condon et al. [19, 20] use PCP techniques to derive new characterizations of PSPACE. A Probabilistically Checkable Debate System (PCDS) for a language L consists of a probabilistic polynomial-time verifier V and a debate between Player 1, who claims that the input x is in L, and Player 0, who claims that the input x is not in L (cf. [19]). An RPCDS is a PCDS in which Player 0 follows a very simple strategy: On each turn, Player 0 chooses uniformly at random from the set of legal moves [20]. A language is in the class PCD($r(n), q(n)$) (resp. RPCD($r(n), q(n)$)) if it has a PCDS (resp. RPCDS) in which V flips $O(r(n))$ random coins and reads $O(q(n))$ bits of the debate. It is shown in [19, 20] that both PCD($\log(n), 1$) and RPCD($\log(n), 1$) are equal to PSPACE.

5.2 Application to Nonapproximability of Optimization Functions

We now show how the PCP Theorem is used to prove nonapproximability results for NP-hard optimization functions. Because approximability of hard optimization functions is a core concern of theoretical computer science, these results demonstrate decisively that coding theory provides useful techniques for mainstream theoretical computer scientists.

The connection between proof systems and hardness of approximation was first drawn by Condon [17]. Feige et al. [24] were the first to connect proof systems to a well-known, basic optimization problem, namely the clique number function defined in Section 2 above. This connection ignited the flurry of work on probabilistically checkable proofs that culminated in the PCP Theorem described in Section 5.1. Here we present a proof that the PCP Theorem implies that MAX-SNP-hard optimization functions do not have PTASs unless P = NP; this version of the proof is taken from Sudan [57].

We begin by defining an optimization function MAX-PCP whose domain consists of pairs (x, L), where L is a language in NP and x is an input, say of size n, for which one may want to determine membership in L. The language L is represented by the description of a PCP($\log(n), 1$) proof system for L. Note that such a proof system must exist by the PCP Theorem and that its description has size $O(1)$. The question of whether x is in L can be transformed into an optimization question as follows. Proof strings in this PCP($\log(n), 1$) system have polynomial length, say $s(n)$. Each bit of a proof string π is regarded as a boolean variable π_i. The verifier in the proof system tosses a sequence of coins that has length $c\log(n)$, for some constant c. Thus there are polynomially many possible coin-toss sequences r. Let $C_{x,r}$ be a constraint on a subset of the variables $\pi_1, \ldots, \pi_{s(n)}$ that evaluates to 1 if the verifier accepts x when it tosses sequence r and evaluates to 0 otherwise. Because the verifier in a PCP($\log(n), 1$) proof system only reads $O(1)$ bits of the proof π, the arity of the constraints $C_{x,r}$ is a constant. The value of the MAX-PCP optimization function on input (x, L) is thus the maximum, over $\pi \in \{0,1\}^{s(n)}$, of the number of simultaneously satisfied constraints $C_{x,r}$.

Now we argue that (1/10)-approximating MAX-PCP is NP-hard. This follows directly from the existence of a "gap" in acceptance probabilities in the definition of a probabilistically checkable proof system. If $x \in L$, then MAX-PCP$(x, L) = n^c$, i.e., there is some proof string π for which the verifier accepts on all coin-toss sequences r, and hence all constraints can be satisfied simultaneously. On the other hand, if $x \notin L$, then MAX-PCP$(x, L) \leq .5n^c$, because, for all proof strings π, the verifier accepts on at most 1/2 the coin-toss sequences r, and hence at most 1/2 of the constraints are simultaneously satisfied. A (1/10)-approximation algorithm for MAX-PCP would thus always return a value that was at least $(1/1.1)n^c \approx .9n^c$ or at most $.55n^c$, thus allowing us to distinguish between the cases $x \in L$ and $x \notin L$. Because L is an arbitrary NP language, this implies that (1/10)-approximating MAX-PCP is NP-hard.

Finally, note that MAX-PCP is in the class MAX-SNP, because it is a constant-arity constraint-satisfaction problem. This means that there is a linear reduction from MAX-PCP to any optimization problem that is MAX-SNP-hard. Thus, any such problem is NP-hard to ϵ-approximate, for some constant ϵ. This means that no such problem has a PTAS, unless P = NP.

The PCP Theorem has been used to derive nonapproximability results for many natural optimization functions, including chromatic number, clique number, MAX-3SAT, shortest vector, nearest vector, and halfspace learning. See [1, 57] for a thorough discussion of these applications. Similarly, the results of Condon et al. [19, 20] are used to derive nonapproximability results for PSPACE-hard optimization functions, including finite-automaton intersection, MAX-Quatified-3SAT, dynamic graph reliability, and games such as generalized geography and mahjongg. See [19, 20] for details.

6 Concluding Remarks

We conclude with pointers to several other connections between coding theory and complexity.

All of the material covered in Sections 3, 4, and 5 deals with the Turing machine model of computation and hence with uniform complexity measures. Coding theory is also used in the study of circuits, formulae, and other nonuniform complexity models. Pippenger [45] provides a thorough overview of this subject.

We have also restricted consideration to results in complexity that make use of or are inspired by coding theory. In fact, the interplay between the two subjects has led to new developments in coding theory as well. For example, Spielman [56] has recently designed an infinite family of codes with constant rate and constant relative distance that can be both encoded and decoded in linear sequential time or logarithmic parallel time with a linear number of processors. Earlier, the "expander codes" of Sipser and Spielman [54] achieved linear-time decoding and quadratic-time encoding algorithms, using the theory of "expander graphs" that is central to many results in complexity theory; the quadratic encoding time makes expander codes inappropriate for on-line communications, but they may be useful for storage on write-once media, because of their efficient decoding algorithms. Both of these recent developments were inspired by the influential role that codes play in the theory of probabilistically checkable proof systems.

In a forthcoming paper, Arora [2] explores this role in depth. His work formalizes the notion of a "code-like reduction," observes that the nonapproximability results discussed in Section 5.2 use such reductions, and shows that these reductions have certain limitations.

References

[1] S. Arora, *Probabilistic Checking of Proofs and Hardness of Approximation Problems*, PhD Thesis, University of California, Computer Science Division, Berkeley CA, 1994.

[2] S. Arora, *Reductions, Codes, PCPs, and Inapproximability*, manuscript, December 1994.

[3] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy, *Proof Verification and Hardness of Approximation Problems*, Proc. 33rd Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, 1992, pp. 14–23.

[4] S. Arora and M. Safra, *Probabilistic Checking of Proofs*, Proc. 33rd Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, 1992, pp. 2–13.

[5] L. Babai, L. Fortnow, L. Levin, and M. Szegedy, *Checking Computations in Polylogarithmic Time*, Proc. 23rd Symposium on Theory of Computing, ACM, New York, 1991, pp. 21–31.

[6] L. Babai, L. Fortnow, and C. Lund, *Nondeterministic Exponential Time has Two-prover Interactive Protocols*, Computational Complexity, 1 (1991), pp. 3–40.

[7] D. Beaver and J. Feigenbaum, *Hiding Instances in Multioracle Queries*, Proc. 5th Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science, vol. 415, Springer, Berlin, 1990, pp. 37–48.

[8] D. Beaver, J. Feigenbaum, J. Kilian, and P. Rogaway, *Security with Low Communication Overhead*, in Advances in Cryptology – Crypto '90, Lecture Notes in Computer Science, vol. 537, Springer, Berlin, 1991, pp. 62–76.

[9] M. Ben-Or, S. Goldwasser, J. Kilian, and A. Wigderson, *Multiprover Interactive Proof Systems: How to Remove Intractability Assumptions*, Proc. 20th Symposium on Theory of Computing, ACM, New York, 1988, pp. 113–131.

[10] M. Ben-Or, S. Goldwasser, and A. Wigderson, *Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation*, Proc. 20th Symposium on Theory of Computing, ACM, New York, 1988, pp. 1–10.

[11] E. Berlekamp and L. Welch, *Error Correction of Algebraic Block Codes*, US Patent Number 4,633,470, 1986. Proof appears in [28].

[12] M. Blum, *Program Result Checking: A New Approach to Making Programs More Reliable*, Proc. 20th International Colloquium on Automata, Languages, and Programming, Lecture Notes in Computer Science, vol. 700, Springer, Berlin, 1993, pp. 2–14. First appeared in preliminary form in International Computer Science Institute Technical Report 88-009, Berkeley CA, 1988.

[13] M. Blum and S. Kannan, *Designing Programs that Check their Work*, J. ACM, to appear. Extended abstract in Proc. 21st Symposium on the Theory of Computing, ACM, New York, 1989, pp. 86–97.

[14] M. Blum, M. Luby, and R. Rubinfeld, *Self-testing/correcting with Applications to Numerical Problems*, J. Comput. Sys. Scis., 47 (1993), pp. 549–595.

[15] R. Boppana, J. Hastad, and S. Zachos, *Does co-NP Have Short Interactive Proofs?*, Inf. Proc. Letters, 25 (1987), pp. 127–133.

[16] D. Chaum, C. Crepeau, and I. Damgard, *Multiparty Unconditionally Secure Protocols*, Proc. 20th Symposium on Theory of Computing, ACM, New York, 1988, pp. 11–19.

[17] A. Condon, *The Complexity of the Max Word Problem and the Power of One-Way Interactive Proof Systems*, Computational Complexity, 3 (1993), pp. 292–305. First appeared in preliminary form in Proc. 8th Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science, vol. 480, Springer, Berlin, 1991, pp. 456–465.

[18] A. Condon, J. Feigenbaum, C. Lund, and P. Shor, *Probabilistically Checkable Debate Systems*, Chicago J. Theoretical Computer Science, to appear. Extended abstract in Proc. 25th Symposium on Theory of Computing, ACM, New York, 1993, pp. 305–314.

[19] A. Condon, J. Feigenbaum, C. Lund, and P. Shor, *Random Debaters and the Hardness of Approximation Stochastic Functions*, SIAM J. Comput., to appear. Extended abstract in Proc. 9th Structure in Complexity Theory Conference, IEEE Computer Society Press, Los Alamitos, 1994, pp. 280–293.

[20] S. Cook, *The Complexity of Theorem-Proving Procedures*, in Proc. 3rd Symposium on the Theory of Computing, ACM, New York, 1971, pp. 151–158.

[21] Van der Waerden, *Algebra*, vol. 1, Frederick Ungar Publishing, New York, 1970.

[22] C. Dwork and M. Naor, *An Efficient Existentially Unforgeable Signature Scheme and its Application*, to appear. Extended abstract in Advances in Cryptology – Crypto '94, Lecture Notes in Computer Science, vol. 839, Springer, Berlin, 1994, pp. 234–246.

[23] S. Even, O. Goldreich, and S. Micali, *On-Line/Off-Line Digital Signatures*, J. Cryptology, to appear. Extended abstract in Advances in Cryptology – Crypto '89, Lecture Notes in Computer Science, vol. 435, Springer, Berlin, 1990, pp. 263–275.

[24] U. Feige, S. Goldwasser, L. Lovász, M. Safra, and M. Szegedy, *Approximating Clique is Almost NP-Complete*, Proc. 32nd Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, 1991, pp. 2–12.

[25] J. Feigenbaum, *Locally Random Reductions in Interactive Complexity Theory*, in Advances in Computational Complexity Theory, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 13, J.-y. Cai (ed.), AMS, Providence, 1993, pp. 73–97.

[26] R. Freivalds, *Fast Probabilistic Algorithms*, in Mathematical Foundations of Computer Science, Lecture Notes in Computer Science, vol. 74, Springer, Berlin, 1979, pp. 57–69.

[27] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of* NP-*Completeness*, Freeman, San Francisco, 1979.

[28] P. Gemmell and M. Sudan, *Highly Resilient Correctors for Polynomials*, Inf. Proc. Letters, 43 (1992), pp. 169–174.

[29] J. Hastad, *Pseudo-Random Generators under Uniform Assumptions*, Proc. 22nd Symposium on the Theory of Computing, ACM, New York, 1990, pp. 395–404.

[30] R. Impagliazzo, L. Levin, and M. Luby, *Pseudo-Random Generation from One-Way Functions*, Proc. 21st Symposium on the Theory of Computing, ACM, New York, 1989, pp. 12–24.

[31] D. Johnson, *A Catalog of Complexity Classes*, in Handbook of Theoretical Computer Science, vol. A: Algorithms and Complexity, J. van Leeuwen (ed.), The MIT Press/Elsevier, Cambridge/New York, 1990, pp. 67–162.

[32] J. Justesen, *A Class of Asymptotically Good Algebraic Codes*, IEEE Trans. Inf. Th., 18 (1972), pp. 652–656.

[33] S. Kannan, *Program Checkers for Algebraic Problems*, PhD Thesis, University of California, Computer Science Division, Berkeley CA, 1989.

[34] R. Karp, *Reducibility Among Combinatorial Problems*, in Complexity of Computer Computations, R. Miller and J. Thatcher (eds.), Plenum, New York, 1972, pp. 85–103.

[35] R. Karp and N. Pippenger, *A Time-Randomness Tradeoff*, talk at the AMS Conference on Probabilistic Computation and Complexity, 1983.

[36] L. Levin, *Universal Sorting Problems*, Problemy Peredaci Informacii, 9 (1973), pp. 115-115 (in Russian). English translation in Problems of Information Transmission, 9, pp. 265–266.

[37] C. Lund, L. Fortnow, H. Karloff, and N. Nisan, *Algebraic Methods for Interactive Proof Systems*, J. ACM, 39 (1992), pp. 859–868.

[38] C. Lund and M. Yannakakis, *On the Hardness of Approximating Minimization Problems*, J. ACM, 41 (1994), pp. 960–981.

[39] J. MacWilliams and N. Sloane, *The Theory of Error-Correcting Codes*, North-Holland, Amsterdam, 1977.

[40] R. McEliece, *The Theory of Information and Coding*, vol. 3 of the Encyclopedia of Mathematics and its Applications, G. Rota (ed.), Addison Wesley, Reading, 1977.

[41] M. Naor, *Bit Commitment Using Pseudorandomness*, J. Cryptology, 4 (1991), 151-158.

[42] J. Naor and M. Naor, *Small-bias Probability Spaces: Efficient Constructions and Applications*, SIAM J. Comput., 22 (1993), pp. 838–856.

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

[44] C. Papadimitriou and M. Yannakakis, *Optimization, Approximation, and Complexity Classes*, J. Comput. Sys. Scis., 43 (1991), pp. 425–440.

[45] N. Pippenger, *Developments in the Synthesis of Reliable Organisms from Unreliable Components*, Proc. AMS Symposia in Pure Maths., 50 (1990), pp. 311–324.

[46] A. Polishchuk and D. Spielman, *Nearly-linear Size Holographic Proofs*, Proc. 26th Symposium on Theory of Computing, ACM, New York, 1994, pp. 194–203.

[47] R. Rubinfeld, *A Mathematical Theory of Self-Checking, Self-Testing, and Self-Correcting Programs*, PhD Thesis, University of California, Computer Science Division, Berkeley CA, 1990.

[48] R. Rubinfeld, *On the Robustness of Functional Equations*, Proc. 35th Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, 1994, pp. 288–299.

[49] R. Rubinfeld and M. Sudan, *Robust Characterization of Polynomials with Applications to Program Testing*, SIAM J. Comput., to appear. Extended abstract in Proc. 3rd Symposium on Discrete Algorithms, ACM/SIAM, New York/Philadelphia, 1992, pp. 23–43.

[50] U. Schoening, *Graph Isomorphisms is in the Low Hierarchy*, in Proc. 4th Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science, vol. 247, Springer, Berlin, 1986, pp. 37–48.

[51] J. Schwartz, *Fast Probabilistic Algorithms for Verification of Polynomial Identities*, J. ACM, 27 (1980), pp. 701–717.

[52] A. Shamir, *How to Share a Secret*, C. ACM, 22 (1979), pp. 612–613.

[53] A. Shamir, IP = PSPACE, J. ACM, 39 (1992), pp. 869–877.

[54] M. Sipser and D. Spielman, *Expander Codes*, Proc. 35th Symposium on Foundations of Computer Science, IEEE Computer Society, Los Alamitos, 1994, pp. 566–576.

[55] N. Sloane, *Codes (Spherical) and Designs (Experimental)*, these proceedings.

[56] D. Spielman, *Linear-Time Encodable and Decodable Error-Correcting Codes*, in Proc. 27th Symposium on Theory of Computing, ACM, New York, 1995, to appear.

[57] M. Sudan, *Efficient Checking of Polynomials and Proofs and the Hardness of Approximation Problems*, PhD Thesis, University of California, Computer Science Division, Berkeley CA, 1992.

[58] S. Zachos, *Probabilistic Quantifiers and Games*, J. Comput. Sys. Sci., 36 (1988), pp. 433–451.