List Decoding in Average-Case Complexity and Pseudorandomness

Venkatesan Guruswami
Department of Computer Science and Engineering
University of Washington
Seattle, WA, U.S.A.
Email: venkat@cs.washington.edu

Abstract— **This is a brief survey into the applications of list decoding in complexity theory, specifically in relating the worst-case and average-case complexity of computational problems, and in construction of pseudorandom generators. Since we do not have space for full proofs, the aim is to give a flavor of the utility of list decoding in these settings together with pointers to where further details can be found.**

I. Introduction

List decoding has been a subject of intense research in recent years. Under list decoding, given a received word the goal is to output all codewords that lie within (Hamming) distance e from it where e is the bound on errors we wish to correct. The flexibility to output a list of codewords instead of a unique one permits one to correct beyond the packing radius (half-the-distance) of the code, while still dealing with a worst-case model of errors. In fact, over large alphabets, such as for Reed-Solomon codes, one can perform efficient list decoding for noise rates close to 100%, while maintaining constant rate (which tends to 0 as the error fraction approaches 1) and outputting only $O(1)$ codewords in the worst-case. This ability to perform error correction for such overwhelming noise rates (when most symbols are in error!) has found numerous applications in complexity theory [9], [12], [2, Chap. 12].

Here we sketch two such applications. At a very informal level, the ability to decode a fully correct codeword from a received word that is almost fully erroneous, corresponds to being able to compute a function correctly everywhere on its domain (i.e., in the worst-case) given access to a noisy oracle that computes the function correctly only on the average (on a very small fraction of the domain). As a result, list decoding has found powerful applications in average-case complexity, and in relating the worst-case hardness to average-case hardness for certain important functions and complexity classes. The first application we sketch is a clean example that illustrates this general paradigm for the special case of the permanent function. Using list decoding, we will show that if we can efficiently compute the permanent on the average for an inverse polynomial fraction of random matrices, then we can compute the permanent for every matrix. This result originally appeared in [1]. In addition to being interesting in its own right, average-case complexity also has further applications to constructing pseudorandom generators and derandomization (i.e., simulation of randomized algorithms by equivalent deterministic ones).

The second application is a lot more involved [7] and we only provide a peek into its proof. It shows how list decoding can be used to build pseudorandom generators *directly* from worst-case hardness (without going through average-case hardness), thus providing a fundamental link between hardness and randomness. Based on a strong enough, but plausible, worst-case hardness assumption, the pseudorandom generators are powerful enough to collapse randomized polynomial time to deterministic polynomial time, i.e., they imply BPP $=$ P. The high level methodology in this construction is as follows. The truth table of the hard function is viewed as a message and encoded by an algebraic list-decodable code, and the pseudorandom generator is based on this encoding. Any small circuit that "breaks" the pseudorandom generator (i.e., distinguishes its output from a random string) is used, together with list decoding algorithms for Reed-Solomon codes, to give a small circuit that computes any desired bit of the message, or equivalently the hard function on every input, thereby contradicting the assumed (worst-case) hardness of the function.

II. Relating Worst-case and Average-case Complexity of the Permanent

The permanent of an $n \times n$ matrix $A = \{A_{ij}\}$ is defined as $\mathsf{perm}(A) = \sum_\sigma \prod_{i=1}^n A_{i\sigma(i)}$ where the summation is over all permutations σ of n elements. Computing the permanent of a given matrix is a fundamental problem that occupies a special place in complexity theory. Valiant showed in 1979 that the problem is $\#P$-hard even for $0/1$ matrices (and is thus no easier than computing the number of satisfying assignments to a polynomial size Boolean formula).

This is a worst-case hardness result, showing no efficient algorithm that correctly computes the permanent of all matrices is likely to exist. Let $M_{p,n}$ denote the set of all $n \times n$ matrices with entries in $\{0, 1, 2, \ldots, p-1\}$ where $p = p(n)$ is a large enough prime.

We are interested in the question: could there exist an algorithm that computes the permanent correctly for a large fraction of matrices in $M_{p,n}$? The following elegant, simple argument due to Lipton [5] (called random self reduction) is the key to relating the worst and average case complexities of the permanent. Let B be a matrix drawn uniformly at random from $M_{n,p}$. Consider the matrix $A + Bx$ where x is an indeterminate. Clearly, the permanent of $A + Bx$ is a univariate polynomial in x of degree at most n; denote $P(x) = \mathsf{perm}(A + Bx)$. Now for each $i \in \{1, 2, \ldots, p\}$, $A+Bi$ is distributed uniformly in $M_{n,p}$. Therefore, if we have an algorithm that computes the permanent correctly for all but a $1/(3(n + 1))$ fraction of matrices in $M_{n,p}$, then running it on input $A + Bi$ for $1 \le i \le n + 1$ will give us the correct value of $P(i)$ for every i in the range $1 \le i \le n + 1$ with probability at least $2/3$. Since $P(x)$ has degree at most n, these $(n+1)$ values suffice to identify the polynomial P and hence $P(0) = \mathsf{perm}(A)$. The last step amounts to erasure decoding of Reed-Solomon codes via polynomial interpolation. By employing error-correction algorithms for Reed-Solomon codes that correct up to half the distance (plus using a different space of matrices that is pairwise independent), one can improve the above connection and show that computing the permanent correctly on $(1/2 + 1/n)$ fraction of matrices in $M_{n,p}$ is as hard as computing it in the worst-case. The factor $1/2$ arises since unique decoding algorithms can never correct more than a fraction $1/2$ of errors.

Since list decoding permits decoding from error rates close to 1, it is conceivable that the average-case connection can be improved to $1/\text{poly}(n)$ fraction of the matrices using list decoding. However, one faces the problem that list decoding may provide only a list of values for the permanent whereas we need to determine the unique correct value. Below, we see how this is handled by using a special "downward self reducibility" property of the permanent which in particular lets us compute the permanent of $n\times n$ matrix based on permanents of $(n-1) \times (n-1)$ matrices.

Suppose for some constant $k > 0$, we have an algorithm $\mathcal{A}$ that computes $\mathsf{perm}(M)$ correctly for a fraction $\gamma = 1/n^k$ of matrices in $M_{p,n}$ for all n and all large enough p, specifically for $p > 10n^{2k+2}$. Let us call any M for which A correctly computes $\mathsf{perm}(M)$ as being *good*.

Let A be the input $n \times n$ matrix whose permanent we wish to compute. Let $A^{(ij)}$ denote the $(n-1) \times (n-1)$ minor of A obtained by deleting the i'th row and j'th column. Let B, C be $(n-1) \times (n-1)$ matrices chosen uniformly and independently at random from $M_{p,n-1}$. Consider the matrix polynomial

$$D(x) = \sum_{i=1}^{n} \delta_i(x) A^{(i1)} + \Big(\prod_{i=1}^{n}(x-i)\Big)(B + xC) \ ,$$

where each $\delta_i(x)$ is a polynomial of degree $(n-1)$ that has value 1 at $x = i$ and vanishes at $x \neq i$ and $1 \le x \le n$. Note that $D(i) = A^{(i1)}$ for $1 \le i \le n$. Denote by $P(x)$ the permanent of $D(x)$; $P(x)$ is a polynomial of degree less than n^2. Clearly, we can compute $\mathsf{perm}(A)$ if we have the correct values of $P(1), P(2), \ldots, P(n)$, and therefore certainly if we knew the polynomial $P(x)$ correctly.

For simplicity, let us assume that p is also upper bounded by a polynomial in n (this is not necessary, but avoids some unnecessary technical issues). Let us run the algorithm $\mathcal{A}$ on the $(p - n)$ matrices $D(j)$ for $n + 1 \le j < p$, obtaining (possibly incorrect) values y_j for their permanent; this can be accomplished in polynomial time due to the bound on p. For each such j, $D(j)$ is a random matrix in $M_{p,n-1}$, and hence it is good with probability $\gamma = 1/n^k$. Moreover, for $j_1 \neq j_2$, the random variables $D(j_1)$ and $D(j_2)$ are independent. Therefore, $\{D(j) \mid n+1 \le j < p\}$ is a pairwise independent collection of random matrices. By a standard Chebyshev inequality argument, with probability close to 1, at least $\gamma/2$ fraction of the matrices in this collection are good, and thus $\mathcal{A}$ outputs the correct value of the permanent.

Thus, among the pairs (j, y_j), $j = n+1, n+2, \ldots, p-1$, the polynomial P "agrees" with at least $(\gamma/2) \cdot (p - n)$ of them, i.e., satisfies $y_j = P(j)$ for at least so many values of j. The goal is to find P, which has degree less than n^2. If $\gamma(p-n)/2 > \sqrt{n^2 \cdot (p-n)}$, then the list decoding algorithm for Reed-Solomon codes from [8], [3] can find all polynomials that agree on at least $\gamma(p-n)/2$ pairs. The above condition is met if $p > 10n^{2k+2}$. Moreover, the size L of the list $\mathcal{L}$ output by the algorithm satisfies $L \le 3/\gamma \le 3n^k$, and this list will include $P(x)$. We are now left with the problem of identifying the correct polynomial $P(x)$ from this list.

Since all polynomials in the list $\mathcal{L}$ have degree less than n^2, a trivial counting argument implies that there are at most $9n^{2k}$ values in $\{0, 1, \ldots, p-1\}$ at which some two polynomials have the same evaluation. It follows that we can find, in deterministic polynomial time (say, by a simple brute-force search), a value $v \in \{0, 1, \ldots, p-1\}$ such that the values $\{Q(v)\}_{Q\in\mathcal{L}}$ are all distinct. Therefore, knowing the value of $P(v)$ suffices to identify P.

But $P(v) = \mathsf{perm}(D(v))$, and therefore to compute $P(v)$ all we need to do is to compute the permanent of a *single* $(n-1) \times (n-1)$ matrix in $M_{p,n-1}$. We can accomplish this task recursively, by doing the exact same process we did above, this time for an $(n-1) \times (n-1)$ matrix (decomposing it into $(n-2) \times (n-2)$ minors, etc.)!

Finally, a comment about the error probability. With suitable choice of parameters, we can ensure the error probability in the recursive step for $n \times n$ matrices is at most $1/(100n^2)$. Therefore, applying this till the size of the matrix becomes some small constant B, incurs an error probability of $\sum_{j=B}^{n} 1/(100j^2) < 1/3$.

In summary, given access to an algorithm that succeeds on a $1/n^k$ fraction of matrices in $M_{n,p}$, we can compute the permanent correctly in the worst-case in randomized polynomial time. We want to stress the crucial use of list decoding of Reed-Solomon codes for noise rates approaching 100% in the above argument.

Extreme average-case hardness from worst-case hardness: We also point the reader to the work [10] where a very strong average-case hardness is established starting from worst-case hardness. Assuming that no circuit of size $2^{\delta n}$ can correctly compute some function $f : \{0,1\}^n \to \{0,1\}$ in the worst-case, a related function f' on $n' = O(n)$ bits is presented such that f' cannot be computed correctly even on a $1/2 + 2^{-\delta' n'}$ fraction of the inputs by circuits of size $2^{\delta' n'}$. Note that getting agreement $1/2$ with f' is trivial — one of the two constant functions 0 or 1 does the job. The result shows that even a tiny bit of advantage over this trivial bound is hard to get. In turn, such an extremely hard-on-average function can be used to build pseudorandom generators that stretch $O(\log n)$ bits to n bits and thereby establish P = BPP [6]. The construction in [10] makes crucial use of error-correcting codes, specifically Reed-Muller codes and a highly efficient list decoding algorithm for it, which in turn uses, among other things, the Reed-Solomon list decoding algorithm as an important subroutine. In the next section, we discuss work that constructs pseudorandom generators directly from worst-case hard functions without the intermediate hardness amplification step. This result again heavily borrows on tools from coding theory and list decoding.

III. Pseudorandom generators from worst-case hardness

We now describe the pseudorandom generator construction by Shaltiel and Umans [7]. This work uses list decoding in a delicate and clever combination with several other involved ideas from algebra and probability. Therefore, we will only be able to provide an informal and incomplete peek into this application.

We now define precisely the notion of a pseudorandom generator. A pseudorandom generator $G : \{0,1\}^s \to \{0,1\}^N$ maps a short seed of length s into a longer string of length N such that no circuit of size N can distinguish (with a non-negligible advantage) between a truly random string from $\{0,1\}^N$ and the output of G on a uniformly random input from $\{0,1\}^s$. The shorter the seed of the generator, the better it is. If we achieve $s = O(\log N)$, then the generator can be used to derandomize polynomial time randomized algorithms with only a polynomial slow down, thus establishing BPP = P. Indeed, this can be done by simply running the randomized algorithm using the output of G on each of the polynomially many seeds in place of the N-bit random string it needs, and then taking a majority vote over all such runs. Assuming the randomized algorithm has circuit complexity at most N, its inability to break the pseudorandom generator implies that it has essentially the same success probability when its random choices are distributed according to the output distribution of G on a random seed as it does on a uniformly random string. Therefore, the above majority voting scheme will yield the correct answer.

The construction of the Shaltiel-Umans pseudorandom generator begins with a hard function $x : \{0,1\}^{\log n} \to \{0,1\}$ (which has n bits of information). Let us assume that no circuit of size at most $n^{0.01}$ can compute x in the worst-case (note that as a function of the length of the input to x (i.e., $\log n$), this is an exponential lower bound). Given such an x, we will present a pseudorandom generator as above with seed length logarithmic in the output length. The pseudorandom generator is based on an encoding of x by an algebraic error-correcting code similar to the Reed-Muller code, except with some additional properties.

We begin with some notation. We denote by $[n]$ the set $\{0, 1, \ldots, n-1\}$. Let $\mathbb{F}$ be a field of size q, and H a subfield of $\mathbb{F}$ of size h (for a suitable $h = n^{\Omega(1)}$, and $q < h^{O(1)}$). Let $d = \Theta(\log n / \log h)$ so that $h^d > n$. We encode x as a low-degree polynomial $\hat{x} : \mathbb{F}^d \to \mathbb{F}$ such that x is "embedded" within the evaluations of $\hat{x}$ on the subcube H^d of $\mathbb{F}^d$ (since $h^d > n$ there are enough slots in H^d for all n bits of x). In other words, $\hat{x}$ will be any polynomial of degree $h-1$ in each variable such $\hat{x}(\ell(i)) = x(i)$ for each $i \in [n]$ where $\ell : [n] \to H^d$ is an efficiently computable one-one map. (This is therefore a systematic version of the Reed-Muller encoding, and is often referred to as the *low-degree extension* of x.) The parameters will be picked so that the map ℓ can take a particularly simple and convenient form (some non-trivial algebra concerning finite fields is needed to ensure this). Specifically, $\ell(i) = A^{pi} \cdot \vec{1}$ where A is a generator of the (cyclic) multiplicative group of $\mathbb{F}^d$ (when $\mathbb{F}^d$ is viewed as an extension field of $\mathbb{F}$), $\vec{1} \in H^d \subseteq \mathbb{F}^d$ refers to the all-ones vector, and the multiplication takes place using the extension field interpretation of $\mathbb{F}^d$. (The integer p will be such that A^p generates the multiplicative group of H^d, viewed as a field extension of H.)

The initial idea for the pseudorandom generator, which won't quite work but is a step in the right direction, is as follows. We use as seed a random element of $\mathbb{F}^d$; thus the seed is a string of $d \log q$ bits. On input $\vec{v} \in \mathbb{F}^d$, the output of the generator consists of the evaluation of $\hat{x}$ on $m = h$ *consecutive* elements of $\mathbb{F}^d$ obtained by successive multiplication by A (again, the multiplication happens in the extension field view of $\mathbb{F}^d$). Formally,

$$G_x(\vec{v}) = \hat{x}(A^1\vec{v}) \circ \hat{x}(A^2\vec{v}) \circ \cdots \circ \hat{x}(A^m\vec{v}) \ .$$

The above outputs field elements where as we would like to output bits. This is achieving using ideas from *code concatenation*. Let C be the encoding function of a binary code of dimension $\log q$ that has an efficient list decoding algorithm up to a fraction $1/2 - \rho$ of errors where $\rho = 1/m^3$, and moreover has block length $\bar{n} = \left(\frac{\log q}{\rho}\right)^{O(1)}$ (it is well known that such codes exist, cf. [4]). The binary pseudorandom generator $\tilde{G}_x$ now takes as seed both $\vec{v} \in \mathbb{F}^d$ and an index $j \in [\bar{n}]$ and outputs the j'th bits of the encoding (by C) of the field elements output by G_x:

$$\tilde{G}_x(\vec{v}; j) = C(\hat{x}(A^1\vec{v}))_j \circ C(\hat{x}(A^2\vec{v}))_j \circ \cdots \circ C(\hat{x}(A^m\vec{v}))_j \ .$$

Note that the output length $m = h = n^{\Omega(1)}$ and the seed length equals $d \log q + O(\log h + \log logq)$ which can be seen to be $O(\log n)$. By well-established techniques (Yao's next bit predictor lemma plus list decoding the inner binary code), a small circuit that breaks the pseudorandom generator $\tilde{G}_x$ can be used to give a small circuit for a "next-element predictor" function that with probability $\Omega(\rho^3)$ correctly predicts the m'th field element output by G_x based on its first $m-1$

field elements. In other words, for a random point $\vec{v} \in F^d$, the probability that

$$f(\hat{x}(A^1\vec{v}), \hat{x}(A^2\vec{v}), \ldots, \hat{x}(A^{m-1}\vec{v})) = \hat{x}(A^m\vec{v})$$

is at least $\Omega(\rho^3)$. The fundamental idea in this construction is to repeatedly use the next-element predictor f, with an error-correction step following each application to ensure the correctness of its output, to efficiently compute $x(i)$ for any desired i. If we can implement such a procedure with a small enough circuit, this would contradict the assumed hardness of the function x, and thus establish that $\tilde{G}_x$ is a pseudorandom generator.

We now elaborate a bit more on how this idea is implemented. Suppose the goal is to compute $x(i) = \hat{x}(A^{ip}\vec{1})$. We pick a random low-degree curve S_1 through $\mathbb{F}^d$ (given by a degree t polynomial $S_1 : \mathbb{F} \to \mathbb{F}^d$). Let $S_2, \ldots, S_{m-1}$ be the successive translates of the curve by multiplication by A, i.e., $S_i = A^{i-1} \cdot S_1$ for $2 \le i < m$, where $A \cdot S_1$ denotes the multiplication of each point in S_1 by A. Note that each of the S_j's are degree t curves since multiplication by A viewed as a field element of the extension field acts as a linear transformation on the vector space $\mathbb{F}^d$ and so preserves degrees. Note that the restriction of $\hat{x}$ to any degree t curve is a univariate polynomial of degree less than tdh. We assume that the correct evaluations of $\hat{x}$ on $S_1, \ldots, S_{m-1}$ are given to us as "advice" — this is the so-called "non-uniformity" in this construction, and we can assume that the advice is "hardwired" in the final circuit we build to compute x.

The procedure to compute $x(i)$ for the input index i is as follows. Consider the sequence of curves $S_a = A \cdot S_{a-1}$ for $a \ge 2$, and let r be the smallest index for which $\ell(i) \in S_r$. Thus if we knew the values of $\hat{x}$ on S_r, we will also know $\hat{x}(\ell(i)) = x(i)$. We compute the values of $\hat{x}$ on the S_a's in sequence using the next-element predictor f, with the advice providing the initial information needed to kick-start the process. In the first step, we run the next-element predictor f in turn for each $(m-1)$-tuple of points on $S_1, S_2, \ldots, S_{m-1}$ (note that we know the value of $\hat{x}$ on all these curves via the advice). This gives us a prediction for $\hat{x}(\vec{w})$ for each $\vec{w} \in S_m$. By virtue of the quality of the next-element predictor, we know that at least a fraction $\Omega(\rho^3)$ of these predictions are correct, i.e., give the true value of $\hat{x}$ at the respective points. Our goal is to find the univariate polynomial $\hat{x}_{|S_m}$ based on these predictions. Now, suppressing details on the certain conditions our parameters must obey, if we solve an underlying Reed-Solomon list decoding problem with these predictions as received word, we can find a list of all univariate polynomials with agreement at least $\Omega(\rho^3)$ with the predictions. This list will therefore include the restriction $\hat{x}_{|S_m}$ of $\hat{x}$ on the curve S_m.

If we could somehow identify the correct polynomial $\hat{x}_{|S_m}$ from this list, we could continue the above process, predicting the value of $\hat{x}_{|S_{m+1}}$ through f and using list decoding to correct the erroneous predictions and identify $\hat{x}_{|S_{m+1}}$, and so on till we compute $\hat{x}_{|S_r}$. Again, similar to what we saw in the previous permanent application, if we knew the value of the correct polynomial (i.e., $\hat{x}_{|S_m}$) at a random point on the curve S_m, then this can be used to uniquely identify the polynomial from the list (with high probability). Shaltiel and Umans [7] achieve precisely this using a clever idea where instead of one iterative process as above, they run two iterative processes that are interleaved in a carefully chosen way. Each of the iterative processes runs as above, but things are set up so that when a process needs the value of the polynomial at a random point on its current curve, this value has already been calculated by the other process! Therefore, the two processes working in tandem enable picking the correct element from the list for each of the list decoding steps. It goes without saying that several technical details are involved in making this work, and we point the reader to [7] for further details on this aspect.

We now mention the serious (and basic) problem with the above approach. Our goal is to construct a circuit to compute $x(i)$ on input i that has size less than $n^{0.01}$ (so that this would contradict the assumed hardness of x). Now the point $\ell(i)$ may be very far-off from the initial curve S_1, i.e., the index r for which $S_r = A^{r-1}S_1$ contains $\ell(i)$ may be very large, in fact we can have $r \approx q^{d-1} = \Omega(n)$. Thus the above iterative process will run for about n time steps, much more than we can afford.

As shown in [7], the ideas discussed above suffice to argue that the above construction yields a good *randomness extractor* (with x being input from a weak-random source, and $(\vec{v}, j)$ being the random seed); see also [11] for a precursor to this extractor construction. However, to get a pseudorandom generator, further ideas are needed which can substantially speed up the above iterative process to compute $\hat{x}(\ell(i))$.

These further ideas are quite natural. In order to travel from the initial curve to any desired location $\ell(i)$ quickly, we use several generators, each with its own "stride" or "successor function", and then combine them suitably. Specifically, for $0 \le i < d$, define the function $G_x^{(i)} : \mathbb{F}^d \to \mathbb{F}^m$ as follows:

$$G_x^{(i)}(\vec{v}) = \hat{x}(A^{q^i \cdot 1}\vec{v}) \circ \hat{x}(A^{q^i \cdot 2}\vec{v}) \circ \cdots \circ \hat{x}(A^{q^i \cdot m}\vec{v}) \ .$$

In other words, the i'th generator uses multiplication by A^{q^i} to compute successive points. The binary version $\tilde{G}_x^{(i)}$ of the generator $G_x^{(i)}$ is obtained analogously to how $\tilde{G}_x$ was obtained from G_x:

$$\tilde{G}_x^{(i)}(\vec{v}; j) = C(\hat{x}(A^{q^i}\vec{v}))_j \circ C(\hat{x}(A^{q^i \cdot 2}\vec{v}))_j \circ \cdots \circ C(\hat{x}(A^{q^i \cdot m}\vec{v}))_j \ .$$

If none of the $\tilde{G}_x^{(i)}$ for $i = 0, 1, 2, \ldots, d-1$ is a pseudorandom generator, then using a similar reasoning to the above, we will have next-element predictors $f^{(i)} : \mathbb{F}^{m-1} \to \mathbb{F}$ with decent success probability for $G_x^{(i)}$ for *each* $i = 0, 1, \ldots, d-1$.

Suppose the advice contains the value of $\hat{x}$ at the points $A^c \cdot \vec{1}, A^{c+1} \cdot \vec{1}, \ldots, A^{c+m-1} \cdot \vec{1}$ for some constant c. Now for $\ell(i) \in \mathbb{F}^d$, one can travel from the point $A^c \cdot \vec{1}$ (which lies on the first curve S_1) to it quickly, in much smaller than q^d steps, as follows. We know that $\ell(i) = A^{ip}\vec{1}$. Therefore we need to "travel" $b = ip - a \pmod{q^d - 1}$ multiples of A from the curve S_1 to reach curve $A^b S_1$ (which will contain $\ell(i)$). Let $b = \sum_{j=1}^{d-1} b_j q^j$ be the q-ary representation of b. We first use the predictor $f^{(0)}$ for strides of length $q^0 = 1$ to take b_0 steps of size 1. At this stage our current location and $\ell(i)$ agree on the least significant digit in the q-ary representation. We next take use the predictor $f^{(i)}$ to compute $\hat{x}$ at a point that agrees with $\ell(i)$ on the least significant two digits, and so on. In the actual implementation, things are a little more subtle than this since we need to ensure that at each stride we have knowledge of the $m-1$ previous evaluations to kick-start the iterative process with the predictor $f^{(i)}$ for that stride. But this description should suffice to illustrate the main idea. In all, after $O(dmq)$ prediction steps, we would get to the location $\ell(i)$ and compute $x(i) = \hat{x}(\ell(i))$. Note that this is much smaller than the $\Omega(q^d)$ steps needed in the earlier approach (with only one stride).

Thus, the above gives a construction of d generators one of which is in fact a pseudorandom generator with the desired property. However, we do not know which one this might be. This problem has a simple fix: we take the output of all the d generators, on *independent* seeds, and do a bitwise XOR of their outputs. It is not hard to show that this will yield a pseudorandom generator (against circuits of slightly smaller size). Therefore, our final pseudorandom generator construction is the following:

$$G_x^{(\mathrm{final})}(y_0, y_1, \ldots, y_{d-1}) = \tilde{G}_x^{(0)}(y_0) \oplus \cdots \oplus \tilde{G}_x^{(d-1)}(y_{d-1}) \ .$$

This completes a description of the main ideas behind the construction in [7]. We hope that this has both given the reader some flavor of the utility of codes and list decoding in this application, as well as piqued the reader's interest to refer to the original paper [7] for more details.

References

[1] Jin-Yi Cai, A. Pavan, and D. Sivakumar. On the hardness of the permanent. In *Proceedings of the 16th International Symposium on Theoretical Aspects of Computer Science*, March 1999.

[2] V. Guruswami. *List decoding of error-correcting codes*. Springer, Lecture Notes in Computer Science 3282, 2004.

[3] V. Guruswami and M. Sudan. Improved decoding of Reed-Solomon and algebraic-geometric codes. *IEEE Transactions on Information Theory*, 45:1757–1767, 1999.

[4] V. Guruswami and M. Sudan. List decoding algorithms for certain concatenated codes. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing*, pages 181–190, 2000.

[5] R. Lipton. New directions in testing, In *Distributed Computing and Cryptography, volume 2 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 191-202, AMS, 1991.

[6] N. Nisan and A. Wigderson. Hardness vs Randomness. *J. Comput. Syst. Sci.*, 49(2):149–167, 1994.

[7] R. Shaltiel and C. Umans. Simple Extractors for All Min-Entropies and a New Pseudo-Random Generator. *Journal of the ACM*, 52(2): 172-21, 2005.

[8] M. Sudan. Decoding of Reed-Solomon codes beyond the error-correction bound. *Journal of Complexity*, 13(1):180–193, 1997.

[9] M. Sudan. List Decoding: Algorithms and Applications. *SIGACT News*, 31:16–27, 2000.

[10] M. Sudan, L. Trevisan, and S. Vadhan. Pseudorandom generators without the XOR lemma. *Journal of Computer and System Sciences*, 62(2):236–266, March 2001.

[11] A. Ta-Shma, D. Zuckerman, and S. Safra. Extractors from Reed-Muller codes. In *Proceedings of the 42nd Annual Symposium on Foundations of Computer Science*, pages 638–647, 2001.

[12] L. Trevisan. Some Applications of Coding Theory in Computational Complexity. *Quaderni di Matematica*, 13:347–424, 2004.