Using Hard Problems to Create Pseudorandom Generators

By

Noam Nisan

B.S. (Hebrew University, Israel) 1985

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA at BERKELEY

Approved: Richard M Karp 12/16/88
Chair Date
Robert Solovay 12/19/88
Uh Vazirani 12/14/88

****

Using Hard Problems to Create Pseudorandom Generators

by

Noam Nisan

ABSTRACT

This thesis describes two methods of constructing pseudorandom generators from hard problems.

We first give a simple and very general construction of pseudorandom generators. They stretch a short string of truly random bits into a long string that looks random to any algorithm from a complexity class C (eg. $P, NC, PSPACE$,...) using an *arbitrary* function that is hard for C. This construction reveals an *equivalence* between the problems of proving certain lower bounds and of constructing pseudorandom generators.

This construction has many consequences. The most direct one is that efficient deterministic simulation of randomized algorithms is possible under much weaker assumptions than previously known. The efficiency of the simulations depends on the strength of the assumptions, and may achieve $P=BPP$. We believe that our results are very strong evidence that the gap between randomized and deterministic complexity is not large.

Using the known lower bounds for constant depth circuits, our construction yields unconditionally proven pseudorandom generators for constant depth circuits. As an application we characterize the power of NP with a random oracle.

Our second pseudorandom generator stretches short truly random strings to long strings that look random to all *Logspace* machines. This is

proved without relying on any unproven assumptions. Instead, we use lower bounds on the complexity of the following multiparty communication game:

Let $f(x_1, \cdots, x_k)$ be a Boolean function that k parties wish to collaboratively evaluate. The i'th party knows each input argument except x_i; and each party has unlimited computational power. They share a blackboard, viewed by all parties, where they can exchange messages. The objective is to minimize the number of bits written on the board.

We prove lower bounds on the number of bits that need to be written on the board in order to compute a certain function. We then use these bounds to construct a pseudorandom generator for Logspace. As an application we present an explicit construction of universal traversal sequences for general graphs.

Richard M. Karp

Prof. Richard M. Karp
Committee Chairman

To Cindy

Acknowledgments

I would first like to thank my advisor, Richard Karp, who has helped my throughout my research, from his posing the first research problem I ever solved to his careful reading of my thesis. His clarity, impeccable teaching style, and scholarly attitude toward research will always be something to aspire to.

I am grateful to have had the privilege of working closely with Mike Sipser during the year he spent in Berkeley. It is hard to exaggerate the extent to which he has influenced my view of computer science.

While visiting the Hebrew University of Jerusalem, I had the pleasure of working with Avi Wigderson. The research described in chapter two was done with his help, guidance and collaboration. I greatly appreciate his friendship and all that I have learned from him.

The results described in chapter three are part of my joint work with Laci Babai and Mario Szegedy at the University of Chicago. I thank them for their collaboration and hospitality.

I have learned much from the many discussions I have had with Manuel Blum and Umesh Vazirani, and from classes they have taught here. I was first introduced to pseudorandom generators in Manuel's Cryptography class, and I can trace the roots of the work described in chapter two back to a homework problem posed by Umesh in his class on Randomness.

One of the best things about studying theoretical computer science in Berkeley is the great group of students and visitors I have had the pleasure to meet and work with. They have been more than friends. Many, many research discussions with Steven Rudich and Russell Imagliazzo have strongly influenced my thinking. I have also enjoyed the opportunity of working with Amos Fiat, Rajeev Motwani, Moni Naor, and Danny Soroker.

Ronitt Rubenfield deserves special thanks for her comments on this manuscript. I am grateful to all of them as well as to Lisa Hellerstein, Sampath Kannan, Valerie King, Assaf Shuster, David Zuckerman and the rest of the theory crowd, for making the department what it is.

Table of Contents

CHAPTER 1:

Introduction

1. Randomized Complexity

In the last decade randomization became an important tool in the design of algorithms. There are many problems for which efficient randomized algorithms have been given even though no efficient deterministic algorithm is known. This apparently enhanced power which randomization provides has become a major research topic in complexity theory.

For almost any natural complexity class, a corresponding randomized class may be defined (in fact several randomized variants can be defined). In most cases no nontrivial relationship is known between the corresponding randomized and deterministic classes, and in many cases there are some interesting problems known to lie in the randomized class but not known to lie in the corresponding deterministic one. Primality testing is known to lie in randomized polynomial time ([SS], [AH]), but not known to lie in deterministic polynomial time. Construction of a perfect matching in a graph is known to lie in random-NC ([KUW], [MVV]), but not known to lie in NC. Undirected connectivity is known to lie in random-Logspace ([AKLLR]), but not known to lie in Logspace. Graph non-isomorphism is known to lie in AM, the randomized analogue of NP ([GMW]), but not known to lie in NP.

This thesis continues the investigation into the power of randomization, and the relationships between randomized and deterministic complexity classes. The line of attack we pursue is the idea of *emulating* randomness, known as *pseudorandom generation*.

2. Pseudorandom Generators

The major conceptual idea behind pseudorandom generation is that sequences of events may *look* random even though they are not truly random in the information theoretic sense. Sequences may look random to any observer that does not have enough *computational* power to "understand" them.

This revolutionary idea was introduced and formalized in the early '80s by Blum and Micali [BM], who were influenced by Shamir [Sh], and by Yao [Y2]. Blum and Micali and Yao proposed the idea of *pseudorandom generators*, functions which stretch a short string of truly random bits into a long string of bits which looks random to observers having limited computational power.

There are several motivations for research into pseudorandom generators. Perhaps the most broadly stated one is merely getting better insight into the nature of randomness from a *behavioral* point of view. More specifically, pseudorandom generation is perhaps the most general and natural method to reduce or eliminate the randomness required by algorithms. Pseudorandom sequences may replace the random sequences required by algorithms without changing the results in any way. This reduces the number of random bits required for solving the problem. Moreover, the trivial exponential deterministic simulation of randomized

algorithms can now be used to obtain rather fast deterministic simulations.

Another motivation for research into pseudorandom generation is cryptography. A standard element implicit in many cryptographic protocols is to generate messages that do not give any information to your opponent, in other words, messages that appear random to him. Pseudorandom generators are a general framework to study these issues. Indeed many cryptographic protocols may be based on pseudorandom generators ([LR], [ILL]).

Finally, perhaps the practical motivation should be mentioned: in reality random bits are required by many computer programs. Hardly ever are truly random bits supplied by the computer (or even claim to be supplied by some physical means such as Zener diodes). Usually some pseudorandom generator supplies numbers which are hoped to be as good as truly random ones. It is important to know how much this can be really justified.

3. Basic Definitions

Blum and Micali [BM] and Yao [Y2] were the first to define pseudorandom generators. They were concerned with pseudorandom generators that look random to polynomial time Turing machines, or to polynomial size circuits. We will consider the natural extensions and give general definitions of pseudorandom generators that look random to an arbitrary class of machines.

Another way in which we modify the definitions given by Blum-Micali and by Yao is the requirement regarding the running time of the pseudorandom generator. Blum-Micali and Yao defined pseudorandom generators to be computed in polynomial time We will remove this requirement from the definition. Of course, for our pseudorandom generators to be

interesting, we will need to show that they can indeed be computed somewhat efficiently.

The definitions that appear here are generic, and more precise specific definitions for particular complexity classes will appear where we use them.

Blum and Micali and Yao gave competing definitions of what a pseudorandom generator should be. These two definitions really turn out to be equivalent. Yao's definition is perhaps the strongest one imaginable: that the pseudorandom bits will behave just like truly random ones in any way measurable by machines in the class.

Definition : $\{G_n:\{0,1\}^{m(n)}\rightarrow\{0,1\}^n\}$ is called a *pseudorandom* generator for class C if for every algorithm A in C, every polynomial $p(n)$, and for all sufficiently large n:

$$\left|Pr[A(y)=1]-Pr[A(G_n(x))=1]\right| \leq \frac{1}{p(n)}$$

Where x is chosen uniformly at random in $\{0,1\}^{m(n)}$, and y in $\{0,1\}^n$.

The definition given by Blum-Micali seems to be a rather minimalist one: that given any prefix of the pseudorandom sequence, the next bit would look random.

Definition : $\{G_n:\{0,1\}^{m(n)}\rightarrow\{0,1\}^n\}$ passes all class C *prediction tests* if for every algorithm A in C, every $1\leq i\leq n$, every polynomial $p(n)$, and all sufficiently large n:

$$\left|Pr[A(y_1,\cdots y_{i-1})=y_i]-1/2\right| \leq \frac{1}{p(n)}$$

Where y_j is the j'th bit output by G_n, and the probability is taken over a random input to G_n.

It turns out that these two definitions are equivalent [Y2].

Theorem (Yao): G is a pseudorandom generator for class C iff it passes all class C prediction tests.

This fact is extremely helpful, since the usual method of proving that a generator is pseudorandom, is to show that it satisfies the weaker definition, and then to conclude that it has all the nice properties of the stronger one.

4. Previous Work

Most work regarding pseudorandom generators has been directed towards pseudorandom generators for P, polynomial time Turing machines. The first pseudorandom number generator was designed by Blum and Micali [BM]. It is based on the unproven assumption that computation of the "discrete log" function cannot be done in polynomial time. They first showed that, under this assumption, the most significant bit of the discrete log cannot be approximated by any polynomial time computation, and proceeded to give a general scheme to produce many pseudorandom bits using this fact.

Yao [Y2] generalized this construction. He showed how any one-way permutation can be used in place of the discrete log function.

Definition: $f=\{f_n:\{0,1\}^n\rightarrow\{0,1\}^n\}$ is called a one-way permutation if (1) f_n is one-one and onto (2) f can be computed in polynomial time (3) Any polynomial size circuit that attempts to invert f, errs on at least a polynomially large fraction of the inputs.

Yao first showed how to "amplify" the "unpredictability" of condition 3, and obtain a "hard bit", a bit that cannot be predicted at all. This amplification is achieved by taking multiple copies of the hard function on disjoint inputs, and xor-ing them, Yao shows that this operation indeed "amplifies" the unpredictability of the bits. Once a "hard bit" is obtained, a

pseudorandom generator can be designed using the Blum-Micali scheme:

A seed x_0 of size n^ϵ is given as the random input. The one-way permutation is applied to the seed repeatedly, generating a sequence $x_0, f(x_0), f(f(x_0)), \cdots, f^{(n)}(x_0)$. The pseudorandom output of the generator is obtained by extracting the "hard bit" from each string in this sequence. The proof that this is indeed a pseudorandom generator proceeds by showing how a test that this sequence fails can be used to invert the one-way permutation f.

Theorem (Yao): If a one-way permutation exists then for every $\epsilon > 0$ there exists a polynomial time computable pseudorandom generator $G:\{0,1\}^{n^\epsilon} \rightarrow \{0,1\}^n$.

Recently, Impagliazzo, Levin and Luby [ILL] proved that the existence of any one way function (not necessarily a permutation) suffices for the construction of pseudorandom generators. This condition is also necessary for the existence of pseudorandom generators that can be computed in polynomial time. All the pseudorandom generators described so far can indeed be computed in polynomial time.

All the work described so far was concerns generators that pass all *polynomial time* tests; pseudorandom generators for P. Some work has also been done regarding the construction of pseudorandom generators for other complexity classes.

Reif and Tygar [RT] describe a generator that passes all NC tests, and moreover, can itself be computed in NC. This generator is based on the assumption that "inverse mod p" cannot be computed in NC. The main innovation here is showing that for this particular function, the original Blum-Micali-Yao generator can be parallelized.

Ajtai and Wigderson [AW] considered pseudorandom generators for AC^0. They use an ad-hoc construction based on the lower bound methods for constant depth circuits, to construct a pseudorandom generator that passes all AC^0 tests. The significance of this result is that it does not require any unproven assumptions! Instead, it builds upon the known, proven lower bounds for constant depth circuits.

5. Hardness vs. Randomness

The second chapter of this thesis is devoted to a new general construction of pseudorandom generators for arbitrary complexity classes. This construction overcomes two basic limitations of the known pseudorandom generators:

(1) They require a strong unproven assumption. (the existence of a one-way function, an assumption which is even stronger than $P \neq NP$)

(2) They are sequential, and can not be applied to an arbitrary complexity class. E.g. there is no known construction of pseudorandom generators for NC that is based on a general complexity assumption about NC.

The construction which we propose avoids both problems: It can be applied to any complexity class C, it is based on an arbitrary function which is hard for C, and gives a pseudorandom generator that looks random to the class C. Although our generator can not necessarily be computed in the class C, we will show that it can be computed efficiently enough for our *simulation* purposes.

Perhaps the most important conceptual implication of this construction is that it proves the *equivalence* between the problem of proving lower bounds for the size of circuits approximating functions in EXPTIME, and

the problem of constructing pseudorandom generators which run "sufficiently fast". This should be contrasted with the result of Impagliazzo, Levin and Luby [ILL] showing the equivalence of proving the existence of *one – way* functions and constructing pseudorandom generators which run in polynomial time. Our construction requires much weaker assumptions, but yields less efficient pseudorandom generators. This loss does not have any effect when using pseudorandom generators for the deterministic simulation of randomized algorithms.

This construction has many implications and a large part of the second chapter describes them. We first show that efficient deterministic simulation of randomized algorithms is possible under much weaker assumptions than previously known. The efficiency of the simulation depends on the strength of the assumption, and can be good enough to show P=BPP. Since the assumptions required for our generator are so weak and natural, we believe that this work provides overwhelming evidence that the gap between deterministic and randomized complexity is not large.

We then turn to pseudorandom generators for constant depth circuits. Since lower bounds for constant depth circuits are known (e.g. [Ha]), our construction yields an unconditionally proven pseudorandom generator for constant depth circuits. This generator improves upon the known generator, due to Ajtai and Wigderson [AW], and implies much better deterministic simulation of randomized constant depth circuits.

Our generator for constant depth circuits turns out to have some interesting consequences regarding the power of random oracles for complexity classes in the polynomial time hierarchy. We show that NP with a random oracle is exactly the class AM, solving an open problem of Babai [BaM]. We also show that a random oracle does not add power to the

polynomial time hierarchy.

A new proof is given of the fact that *BPP* is in the polynomial time hierarchy. Our final application is a surprising connection between simulation of "time by space" and simulation of "randomness by determinism". We show that one of these simulations can be substantially improved over known simulations.

The results in this chapter have appeared in [NW1], [NW2], and are joint work with Avi Wigderson.

6. Pseudorandom Generators for Logspace

The third chapter in this thesis describes the construction of a pseudorandom generator for Logspace. This result is unique in that it is not based on any unproven assumptions. So far the only class for which pseudorandom generators were unconditionally proven to exist was AC^0.

In order to prove the correctness of our pseudorandom generator we use the following multiparty communication game, first introduced by Chandra, Furst and Lipton [CFL]: Let $f(x_1, \cdots, x_k)$ be a Boolean function that accepts k arguments each n bits long. k parties wish to collaboratively evaluate f; the i'th party knows each input argument except x_i; and each party has unlimited computational power. They share a blackboard, viewed by all parties, where they can exchange messages. The objective is to minimize the number of bits written on the board.

We first prove lower bounds for the number of bits that must be written on the board in order to get even a small advantage on computing certain functions. We then show how to use these lower bounds in order to construct a pseudorandom generator for Logspace.

We conclude by giving some applications of our pseudorandom generator. We describe a construction of universal sequences for arbitrary regular graphs; no nontrivial such construction was previously known. We also show that random Logspace machines with two-way access to the random bits are better, in some specific sense, than random Logspace machines with the usual one-way access to the random bits.

The results in this chapter have appeared in [BNS] and are joint work with Laszlo Babai and Mario Szegedy.

CHAPTER 2:

Hardness vs. Randomness

1. Introduction

In this chapter we describe a general construction of pseudorandom generators which can be based on the hardness of an arbitrary function, and can be applied to any complexity class.

In section two we define, describe in detail, and prove the correctness of our pseudorandom generator. In section three we then prove a host of applications and corollaries of our construction.

2. The generator

In this section we state and prove our results for pseudorandom generators that look random to small circuits, and thus also to time-bounded Turing machines. All the definitions and theorems we give have natural analogues regarding pseudorandom generators for other complexity classes such as depth-bounded circuits, etc. It is rather straightforward to make the required changes, and we leave it to the interested reader.

2.1. Definitions

Informally speaking, a pseudorandom generator is an "easy to compute" function which converts a "few" random bits to "many" pseudorandom bits that "look random" to any "small" circuit. Each one of the quoted words is really a parameter, and we may get pseudorandom generators of different qualities according to the choice of parameter. For example, the standard definitions are: "easy to compute" = polynomial time; "few" = n^{ε}; "many" = n; "look random" = subpolynomial difference in acceptance probability; and "small" = any polynomial. We wish to present a more general tradeoff, and obtain slightly sharper results than these particular choices of parameters allow. Although all these parameters can be freely traded-off by our results, it will be extremely messy to state everything in its full generality. We will thus restrict ourselves to two parameters that will have multiple purposes. The choice was made to be most natural from the "simulation of randomized algorithms" point of view.

The first parameter we have is "the quality of the output", this will refer to 3 things: the number of bits produced by the generator, the maximum size of the circuit the generator "fools", and the reciprocal of difference in accepting probability allowed. In general, in order to simulate a certain randomized algorithm, we will require a generator with quality of output being approximately the running time of the algorithm.

The second parameter is "the price" of the generator, this will refer to both the number of input bits needed, and to the logarithm of the running time of the generator. In general, the deterministic time required for simulation will be exponential in the "price" of the generator.

Definition: $G=\{G_n:\{0,1\}^{l(n)}\rightarrow\{0,1\}^n\}$, denoted by $G:l\rightarrow n$, is called a *pseudorandom generator* if for any circuit C of size n:

$$\left|Pr\left[C(y)=1\right]-Pr\left[C(G(x))=1\right]\right| < 1/n$$

where y is chosen uniformly in $\{0,1\}^n$, and x in $\{0,1\}^l$.

We say G is a *quick* pseudorandom generator if it runs in deterministic time exponential in its *input* size, $G \in DTIME(2^{O(l)})$.

We will also define an *extender*, a pseudorandom generator that only generates one extra bit:

Definition: $G=\{G_l:\{0,1\}^l \rightarrow \{0,1\}^{l+1}\}$ is called an *n-extender* if for any circuit C, of size n:

$$\left|Pr\left[C(y)=1\right]-Pr\left[C(G(x))=1\right]\right| \leq 1/n$$

where y is chosen uniformly in $\{0,1\}^{l+1}$, and x in $\{0,1\}^l$.

We say G is a *quick* extender if it runs in deterministic time exponential in its *input* size, $G \in DTIME(2^{O(l)})$.

The major difference between our definition, and the "normal" definition is the requirement regarding the running time of the algorithm: normally the pseudorandom generator is required to run in polynomial time, we allow it to run in time exponential in its *input* size. This relaxation allows us to construct pseudorandom generators under much weaker conditions than the ones required for polynomial time pseudorandom generators, but our pseudorandom generators are as good for the purpose of simulating randomized algorithms as polynomial time ones. The following lemma is the natural generalization of Yao's [Y2] lemma showing how to use pseudorandom generators to simulate randomized algorithms:

Lemma 2.1: If there exists a quick pseudorandom generator $G:l(n) \rightarrow n$ then for any time bound $t=t(n)$: $RTIME(t) \subset DTIME(2^{O(l(t^2))})$.

Proof: The simulation can be partitioned into two stages. First, the original randomized algorithm which uses $O(t$ random bits is simulated by a

randomized algorithm which uses $l(t^2)$ random bits but runs in time $2^{O(l(t^2))}$. This is done simply by feeding the original algorithm pseudorandom sequences obtained by the generator instead of truly random bits. Since the output of the pseudorandom generator looks random to any circuit of size t^2, and since any algorithm running in time t can be simulated by a circuit of size t^2, the output of the generator will look random to the original algorithm. Thus the probability of acceptance of this randomized algorithm will be almost the same as of the original one.

In the second stage we simulate this randomized algorithm deterministically, by trying all the possible random seeds and taking a majority vote. The number of different seeds is $2^{l(t^2)}$, and for each one a computation of complexity $2^{O(l(t^2))}$ is done. ■

2.2. Hardness

The assumption under which we construct a generator is the existence of a "hard" function. By "hard" we need not only that the function can not be computed by small circuits but also that it can not be *approximated* by small circuits. There are basically two parameters to consider: the size of the circuit and the closeness of approximation.

Definition: Let $f:\{0,1\}^n \rightarrow \{0,1\}$ be a boolean function. We say that f is (ε,S)-*hard* if for any circuit C of size S,

$$\left| Pr\left[C(x)=f(x)\right] - 1/2 \right| < \varepsilon/2$$

Where x is chosen uniformly at random in $\{0,1\}^n$.

Yao [Y2] shows how the closeness of approximation can be amplified by xor-ing multiple copies of f.

Lemma 2.2 (Yao) : Let $f_1, \cdots, f_k$ all be (ε,S)-hard. Then for any $\delta>0$, the function $f_1(x_1)+\cdots+f_k(x_k)$ is $(\varepsilon^k+\delta,\delta^2(1-\varepsilon)^2S)$-hard. Where "+" denotes exclusive-or, i.e. addition mod 2.

The kind of hardness we will require in our assumption is the following:

Definition: Let $f=\{0,1\}^*\rightarrow\{0,1\}$ be a boolean function. We say that f *cannot be approximated* by circuits of size $s(n)$ if for some constant k, all large enough n, and all circuits C_n of size $s(n)$:

$$Pr\left[C_n(x)\neq f(x)\right]>n^{-k}$$

where x is chosen uniformly in $\{0,1\}^n$.

This is a rather weak requirement, as it only requires that small circuits attempting to compute f have a non-negligible fraction of error. Yao's xor-lemma allows amplification of such hardness to the sort of hardness which we will use in our construction. We will want that that no small circuit can get any non-negligible advantage in computing f.

Definition: Let $f:\{0,1\}^*\rightarrow\{0,1\}$ be a boolean function, and let f_m be the restriction of f to strings of length m. The *Hardness* of f at m, $H_f(m)$ is defined to be the maximum integer h_m such that f_m is $(1/h_m,h_m)$-hard.

The following lemma is an immediate application of Yao's lemma.

Corollary 2.3: Let $s(m)$ be any function (size-bound) such that $m\leq s(m)\leq 2^m$; if there exists a function f in EXPTIME that cannot be approximated by circuits of size $s(m)$, then for some $c>0$ there exists a function f' in EXPTIME that has hardness $H_{f'}(m)\geq s(m^c)$.

2.3. The Main Lemma

Given a "hard" function, it is intuitively easy to generate one pseudorandom bit from it since the value of the function must look random to any small circuit. The problem is to generate more than one pseudorandom bit. In order to do this we will compute the function on many different, nearly disjoint subsets of bits.

Definition: A collection of sets $\{S_1, \cdots, S_n\}$, where $S_i \subset \{1, \cdots, l\}$ is called a (k,m)-*design* if:

(1) For all i:

$$\left|S_i\right| = m$$

(2) For all $i \neq j$:

$$\left|S_i \bigcap S_j\right| \leq k$$

A $n \times l$ 0-1 matrix is called a (k,m)-design if its n rows, interpreted as subsets of $\{1..l\}$ are a (k,m)-design.

Definition: Let A be a $n \times l$ 0-1 matrix, let f be a boolean function, and let $x = (x_1 \cdots x_l)$ be a boolean string. Denote by $f_A(x)$ the n bit vector of bits computed by applying the function f to the subsets of the x's denoted by the n different rows of A.

Our generator expands the seed x to the pseudorandom string $f_A(x)$. The quality of the bits is assured by the following lemma.

Lemma 2.4: Let m,n,l be integers; let f a boolean function, $f:\{0,1\}^m \rightarrow \{0,1\}$, such that $H_f(m) \geq n^2$; and let A be a boolean $n \times l$ matrix which is a $(\log n, m)$ design. Then $G:l \rightarrow n$ given by $G(x) = f_A(x)$ is a pseudorandom generator.

Proof: We will assume that G is not a pseudorandom generator and derive a contradiction to the hardness assumption. If G is not a pseudorandom generator then, wlog, for some circuit C, of size n,

$$Pr\left[C(y)=1\right]-Pr\left[C(G(x))=1\right] > 1/n$$

We first show, as in [GM] and in [Y2], that this implies that one of the bits of $f_A(x)$ can be predicted from the previous ones.

For any i, $0 \le i \le n$, we define a distribution E_i on $\{0,1\}^n$ as follows: the first i bits are chosen to be the first i bits of $f_A(x)$, where x is chosen uniformly over l bit strings, and the other $n-i$ bits are chosen uniformly at random. Define

$$p_i = Pr\left[C(z)=1\right]$$

where z is chosen according to the distribution E_i. Since $p_0 - p_n > 1/n$, it is clear that for some i, $p_{i-1} - p_i > 1/n^2$. Using this fact we will build a circuit that predicts the i'th bit.

Define a circuit D, which takes as input the first $i-1$ bits of $f_A(x)$, $y_1, \cdots, y_{i-1}$, and predicts the i'th bit, y_i. D is a probabilistic circuit. It first flips $n-i+1$ random bits, $r_i, \cdots, r_n$. On input $y = <y_1, \cdots, y_{i-1}>$, it computes $C(y_1, \cdots, y_{i-1}, r_i, \cdots, r_n)$. If this evaluates to 1 then D will return r_i as the answer, otherwise it will return the complement of r_i. As in [Y2] it can be shown that

$$Pr\left[D_n(y_1, \cdots, y_{i-1}) = y_i\right] - \frac{1}{2} > \frac{1}{n^2}$$

where the probability is taken over all choices of x and of the random bits that D uses. At this point an averaging argument shows that it is possible to set the private random bits that D uses to constants and achieve a deterministic circuit D' while preserving the bias

By now we have constructed a circuit that predicts y_i from the bits $y_1, \cdots, y_{i-1}$. To achieve a contradiction to the hardness assumption we will now transform this circuit to a circuit that predicts y_i from the bits $x_1, \cdots, x_l$. W.l.o.g. we can assume that y_i depends on $x_1, \cdots, x_m$, i.e.

$$y_i = f(x_1 \cdots x_m)$$

Since y_i does not depend on the other bits of x, it is possible to set the other bits to constants, while leaving the prediction of y_i valid. By an averaging argument there exist constants $c_{m+1}, \cdots, c_l$ such that setting $x_j = c_j$ for all $m < j \leq l$, preserves the prediction probability. At this point, however, each one of the bits $y_1, \cdots, y_{i-1}$ depends only on at most $\log n$ of the bits $x_1, \cdots, x_m$. This is so since the intersection of the set of x_k's defined by y_i and by y_j is bounded from above by $\log n$ for each $i \neq j$. Now we can compute each y_i as a CNF (or DNF) formula of a linear (in n) size over the bits it uses. This gives us a circuit $D''(x_1, \cdots x_m)$ that predicts y_i which is $f(x_1, \cdots, x_m)$. It is easy to check that the size of D'' is at most n^2, and the bias achieved is more than n^{-2}, which contradicts the assumption that $H_f(m) > n^2$. ■

2.4. Construction of Nearly Disjoint Sets

This section describes the actual construction of the designs that are used by the pseudorandom generator. In the construction of the generator, we are given a "hard" function f with a certain "hardness", H_f, and we wish to use it to generate a pseudorandom generator $G: l \rightarrow n$. Our aim is to minimize l, that is to get a pseudorandom generator that uses the smallest number of random bits. If we look at the requirements of lemma 2.4, we see that we will require a $(\log n, m)$-design, where m must satisfy $H_f(m) \geq n^2$.

This basically determines a minimum possible value for m. The following lemma shows that l need not be much larger than m.

Lemma 2.5: For every integers n and m, such that $\log n \leq m \leq n$, there exists an $n \times l$ matrix which is a $(\log n, m)$-design, where $l \leq O(m^2)$. Moreover, the matrix can be computed by a Turing machine running in space $O(\log n)$.

Proof: We need to construct n different subsets of $\{1 \cdots l\}$ of size m with small intersections. Assume, wlog, that m is a prime power, and let $l = m^2$. (If m is not a prime power, pick, e.g., the smallest power of 2 which is greater than m; this can at most double the value of m) Consider the numbers in the range $\{1 \cdots l\}$ as pairs of elements in $GF(m)$, i.e. we construct subsets of $\{<a,b> \mid a,b \in GF(m)\}$. Given any polynomial q on GF(m), we define a set $S_q = \{<a,q(a)> \mid a \in GF(m)\}$. The sets we take are all of this form, where q ranges over polynomials of degree at most $\log n$. The following facts can now be easily verified:

(1) The size of each set is exactly m.

(2) Any two sets intersect in at most $\log n$ points.

(3) There are at least n different sets (the number of polynomials over $GF(m)$ of degree at most $\log n$ is $m^{\log n + 1} \geq n$).

It should be noted that all that is needed to construct these sets effectively is simple arithmetic in $GF(m)$, and since m has a length of $O(\log n)$ bits, everything can be easily computed by a log-space bounded Turing machine. ■

It can be shown that the previous design is optimal up to a factor of $\log n$, i.e for given m and n, l is within a log factor of the design with the smallest value of l. For most values of m this small added factor is not so

important, however for small values of m we may wish to do better. One way to achieve a better design for small values of m is to consider multivariate polynomials over finite fields. These multinomials may define sets in a similar manner as in the previous design, and for small values of m, l can be reduced up to about $m\log m$. We leave the details to the interested reader.

A case of special interest is $m=O(\log n)$. In this case it is possible to reduce l also to $O(\log n)$. We do not have an explicit construction for this, however we note that such a design can be computed in polynomial time.

Lemma 2.6: For every integers n and m, where $m=C\log n$, there exists a $n\times l$ matrix which is a $(\log n,m)$-design where $l=2C^2\log n$. Moreover, the matrix can be computed by a Turing machine running in time polynomial in n.

Proof: The Turing machine will *greedily* choose subsets of $\{1,\cdots,l\}$ of cardinality m, which intersect each of the previously chosen sets at less than $\log n$ points. A simple counting argument shows that it is always possible to choose such a set, whatever the previous sets that were chosen are, as long as there are at most n such sets. The running time is polynomial since we are looking at subsets of $O(\log n)$ elements. ■

2.5. Main Theorem

The main theorem we get is a necessary and sufficient condition for the existence of quick pseudorandom generators.

Theorem 1: For every function (size bound) $l\le s(l)\le 2^l$ the following are equivalent:

(1) For some $c>0$ some function in EXPTIME cannot be approximated by circuits of size $s(l^c)$.

(2) For some $c>0$ there exists a function in EXPTIME with hardness $s(l^c)$.

(3) For some $c>0$ there exists a quick $s(l^c)$-*extender* $G:l\rightarrow l+1$.

(4) For some $c>0$ there exists a quick pseudorandom generator $G:l\rightarrow s(l^c)$.

Proof:

(1) -> (2) is corollary 2.3.

(4) -> (3) is trivial.

(3) -> (1) is proven by the following observation: Let $G=\{G_l\}$ be an extender as in (3). Consider the problem of "Is y in the range of G?". It can be easily seen that this can be computed in exponential time; however, no circuit of size $s(l^c)$ can approximate it since that circuit would distinguish between the output of G and between truly random strings.

The main part of the proof is, of course, (2) -> (4). Let f be a function in EXPTIME with hardness $s(l^c)$. We build a quick pseudorandom generator $G:l\rightarrow n$, for $n=s(m^{c/4})$: For every n let A_n be the matrix guaranteed by lemma 2.5 for $m=l^{1/2}$. Notice that this is an $n\times l$ matrix which is a $(\log n,m)$-design. Notice also that, by our choice of parameters, $H_f(m)>n^2$. Thus, by lemma 2.4, the function $G_n(x)=f_{A_n}(x)$ is a pseudorandom generator. $G=\{G_n\}$ is a quick pseudorandom generator simply since f is in EXPTIME. ■

This theorem should be contrasted with the known results regarding the conditions under which *polynomial time* computable pseudorandom generators exist. Impagliazzo, Levin and Luby [ILL] prove the following theorem:

Theorem ([ILL]): The following are equivalent (for any $1>\varepsilon>0$):

(1) There exists a 1-way function.

(2) There exists a polynomial time computable pseudorandom generator $G:n^{\epsilon}\rightarrow n$.

The existence of polynomial time computable pseudorandom generators seems to be a stronger statement, and requires apparently stronger assumptions than the existence of "quick" pseudorandom generators.

3. Main Corollaries

3.1. Sequential Computation

The major application of the generator is to allow better deterministic simulation of randomized algorithms. We now state the results we get regarding the deterministic simulation of BPP algorithms.

Theorem 2: If there exists a function computable in $DTIME(2^{O(n)})$,

(1) that cannot be approximated by polynomial size circuits. Or,

(2) that cannot be approximated by circuits of size $2^{n^{\epsilon}}$ for some $\epsilon>0$. Or,

(3) with hardness $2^{\epsilon n}$ for some $\epsilon>0$.

Then

(1) $BPP\subset \bigcap_{\epsilon>0} DTIME(2^{n^{\epsilon}})$.

(2) $BPP\subset DTIME(2^{(\log n)^{c}})$ for some constant c.

(3) $BPP=P$.

respectively.

Proof: using theorem 1, (1) implies the existence of a quick pseudorandom generator $G:n^{\epsilon}\rightarrow n$ for every $\epsilon>0$, and (2) implies the existence of a quick pseudorandom generator $G:(\log n)^{c}\rightarrow n$ for some $c>0$. (3) implies the

existence of a quick pseudorandom generator $G:C\log n \to n$ for some $C>0$. This can be seen by modifying the proof of theorem 1 as to use the design specified in lemma 2.6 instead of the "generic" design (lemma 2.5). The simulation results follow by lemma 2.1. ■

3.2. Parallel Computation

The construction of the generator was very general, it only depended on the existence of a function that was hard for the class the generator is intended for. Thus we can get similar simulation results for other complexity classes under the analogous assumptions. We will now state the major simulation results we get for parallel computation.

Theorem 3: If there exists a function in PSPACE that

(1) cannot be approximated by NC circuits. Or

(2) cannot be approximated by circuits of *depth* n^{ε} (for some constant $\varepsilon>0$).

Then

(1) $RNC \subset \bigcap_{\varepsilon>0} DSPACE(n^{\varepsilon})$.

(2) $RNC \subset DSPACE(polylog)$.

Respectively.

Proof: The proof is the straightforward adaptation of our pseudorandom generator to the parallel case. The important point is that the generator itself is parallel, and indeed in the proof of the main lemma, the depth of the circuit C increases only slightly. ■

3.3. Constant Depth Circuits

A special case of interest is the class of constant depth circuits. Since for this class lower bounds are known, we can use our construction to obtain pseudorandom generators for constant depth circuits that do not require any unproven assumption.

Our generator is based on the known lower bounds for constant depth circuits computing the parity function. We will use directly the strongest bounds known due to Hastad [Ha].

Theorem (Hastad): For any family $\{C_n\}$ of circuits of depth d and size at most $2^{n^{\frac{1}{d+1}}}$, and for all large enough n:

$$\left|Pr\left[C_n(x)=parity(x)\right]-1/2\right| \leq 2^{n^{\frac{1}{d+1}}}$$

When x is chosen uniformly over all n-bit strings. ■

Applying to this our construction we get:

Theorem 4: For any integer d, there exists a family of functions: $\{G_n:\{0,1\}^l \rightarrow \{0,1\}^n\}$, where $l=O((\log n)^{2d+6})$ such that:

(1) $\{G_n\}$ can be computed by a log-space uniform family of circuits of polynomial size and $d+4$ depth.

(2) For any family $\{C_n\}$ of circuits of polynomial size and depth d, for any polynomial $p(n)$, and for all large enough n:

$$\left|Pr\left[C_n(y)=1\right] - Pr\left[C_n(G_n(x))=1\right]\right| \leq \frac{1}{p(n)}$$

where y is chosen uniformly in $\{0,1\}^n$, and x is chosen uniformly in $\{0,1\}^l$.

Proof: Again $G_n=f_{A_n}$ where f is the parity function, and A_n is the design described in section 2.3 for $m=(\log n)^{d+3}$. Notice that: (1) the generator can

be computed by polynomial size circuits of depth $d+4$ since it is just the parity of sets of bits of cardinality $(\log n)^{d+3}$. (2) All the considerations in the proof of correctness of the generator apply also to constant depth circuits. In particular the depth of the circuit C in the proof of lemma 2.4 increases only by one. ■

We can now state the simulation results we get for randomized constant depth circuits. Denote by RAC^0 ($BPAC^0$) the set of languages that can be recognized by a uniform family of *Probabilistic* constant depth, polynomial size circuits, with 1-sided error (2-sided error bounded away from 1/2 by some polynomially small fraction).

Theorem 5:

$$BPAC^0 , RAC^0 \subset \bigcup_c DSPACE((\log n)^c)$$

and

$$BPAC^0 , RAC^0 \subset \bigcup_c DTIME(2^{(\log n)^c})$$

■

Denote by #DNF the problem of counting the number of satisfying assignments to a DNF formula, and by Approx-#DNF the problem of computing a number which is within a factor of 2 (or even $1+n^{-k}$) from the correct value. Clearly #DNF is #P complete. However, our results imply that:

Corollary 3.1: Approx-#DNF $\in DTIME(2^{(\log n)^{14}})$

Proof: Karp and Luby [KLu] give a probabilistic algorithm for Approx-#DNF. It is not difficult to see that this algorithm can be implemented by RAC^0 circuits of depth 4. ■

3.4. Random Oracles

The existence of our pseudorandom generator for constant depth circuits has implications concerning the power of random oracles for classes in the polynomial time hierarchy.

Let C be any complexity class (e.g. P, NP, ...). As in [BM] we define the class $almost-C$ to be the set of languages L such that:

$$Pr\left[L \in C^A\right] = 1$$

where A is an oracle chosen at random. The class $almost-C$ can be thought of as a natural probabilistic analogue of the class C.

The following theorem is well known ([Ku], [BG]), and underscores the importance of BPP as the random analogue of P:

Theorem: BPP = almost-P ■

Babai [Ba] introduced the class AM. An AM Turing machine is a machine that may use both randomization and nondeterminism, but in this order only, first flip as many random bits as necessary and then use nondeterminism. The machine is said to accept a language L if for every string in L the probability that there exists an accepting computation is at least 2/3, and for every string not in L the probability is at most 1/3 (the probability is over all random coin flips, and the existence is over all nondeterministic choices). The class AM is the set of languages accepted by some AM machine that runs in polynomial time. The randomization stage of the computation is called the "Arthur" stage and the second stage, the nondeterministic one is called the "Merlin" stage. For exact definitions as well as motivation refer to [Ba], [BaM], also see [GS].

[BaM] and [GS] raised the question of whether AM = almost-NP? This would strengthen the feeling that AM is the probabilistic analogue of NP.

Our results imply that this is indeed the case.

Theorem 6: AM = almost-NP.

Proof: We first show that $AM \subset almost-NP$. Given an AM machine we can first reduce the probability of error such that for a given $\varepsilon > 0$, on any input of length n, the machine errs with probability bounded by $\varepsilon 4^{-n}$. An NP machine equipped with a random oracle can use the oracle to simulate the *Arthur* phase of the AM machine. For any given input, this machine will accept with the same probability as the AM machine. By summing the probabilities of error over all possible inputs we get that the probability that this machine errs on *any* input is at most ε. Since ε is arbitrary we get that $AM \subset almost-NP$.

We will now prove $almost-NP \subset AM$. We first prove the following fact:

Fact: If L $\in$ almost-NP then there exists a specific nondeterministic oracle Turing machine M that runs in polynomial time such that for an oracle A chosen at random:

$$Pr\left[M^A \text{ accepts } L\right] \geq 2/3$$

Proof (of fact): Since there are only countably many Turing machines, some fixed Turing machine accepts the language L on non-zero measure of oracles. By using the Lebesgue density theorem, we see that it is possible to fix some finite prefix of the oracle such that for oracles with this prefix the Turing machine accepts L with probability at least 2/3. Finally, this prefix can be hard-wired into the Turing machine.

Up to this point we have only used the standard tools. The difficulty comes when we try to simulate M (with a random oracle) by an AM machine. The difficulty lies in the fact that the machine may access (non deterministically) an exponential number of locations of the oracle, but AM

computations can only supply a polynomial number of random bits. We will use our generator to convert a polynomial number of random bits to an exponential number of bits that "look" random to the machine M.

Let the running time of M be n^k. We can view the computation of M as a large OR of size 2^{n^k} of all the deterministic polynomial time computations occurring for the different nondeterministic choices. Each of these computations can be converted to a CNF formula of size 2^{n^k} over the oracle entries. Altogether the computation of M can be written as a depth 2 circuit of size at most 2^{2n^k} over the oracle queries.

Our generator can produce from $2n^{10k}$ random bits 2^{2n^k} bits that look random to any depth 2 circuit of this size. So the simulation of M on a random oracle proceeds as follows: Arthur will flip $2n^{10k}$ random bits, and then M will be simulated by Merlin; whenever M makes an oracle query, the answer will be generated from the random bits according to the generator. Note that this is just a parity function of some subset of the bits, which is clearly in P. Since the generator "fools" this circuit, the simulation will accept with approximately the same probability that M accepts on a random oracle. ■

Exactly the same technique suffices to show that for any computation in PH, the polynomial time hierarchy ([St], [CKS]), a random oracle can be substituted by an "Arthur" phase. Applying to this Sipser's [Si1] result that $BPP \subset \Sigma_2 \bigcap \Pi_2$ allows simulation of the "Arthur" phase by one more alternation and thus we get:

Theorem 7: almost-PH = PH ■

3.5. BPP and the Polynomial Time Hierarchy

In [Si1] Sipser showed that BPP could be simulated in the polynomial time hierarchy. Gacs improved this result and showed simulation is possible in $\Sigma_2 \bigcap \Pi_2$. In this section we give a new simple proof of this fact.

Theorem 8 (Sipser, Gacs): $BPP \subset \Sigma_2 \bigcap \Pi_2$.

Proof: It suffices to show that $BPP \subset \Sigma_2$. The main idea is that a pseudorandom generator that stretches $O(\log n)$ random bits to n pseudorandom bits can be constructed in Σ_2. To simulate BPP then, a Σ_2 machine will then run over all of the polynomially many possibilities of the random seed.

To get such a pseudorandom generator, using our construction, we only need a function with exponential hardness (specifically we want a function on $O(\log n)$ bits with hardness which is $\Omega(n^2)$). Such a function can be found in Σ_2: A simple counting argument shows that such a function exists (although non uniformly), and verifying that a function on $O(\log n)$ bits has indeed a high hardness can easily be seen to be in Co-NP. (The function can be described by a polynomial size table, and the verification can be done by nondeterministically trying all circuits of size n^2).

Thus the simulation of BPP will proceed as follows: (1) Nondeterministically guess a function on $O(\log n)$ bits with high hardness (first alternation). (2) Verify it is indeed hard (Second alternation). (3) Use it as a basis for the pseudorandom generator, using our construction. (4) Try all possible seeds. ■

Actually, this proves a slightly stronger statement, namely that $BPP \subset ZPP^{NP}$. (ZPP^{NP} is the class of languages that have polynomial time, randomized, *zero error* algorithms, using an NP-complete oracle).

3.6. Randomness and Time vs. Space

Our generator is based on the assumption that there exists a function in, say, $DTIME(2^n)$, that can not be approximated by small circuits. In this section we show that if this assumption does not hold then some nontrivial simulation of time by space is possible.

This result shows that either randomized algorithms can be simulated deterministically with subexponential penalty, or that, in some sense, an algorithm that runs in time T can be simulated in space $T^{1-\varepsilon}$, for some $\varepsilon>0$. This simulation is significantly better than the best known simulation of time T in space $T/\log T$ due to Hopcroft, Paul and Valiant [HPV]. A result of a similar flavor, giving a tradeoff between simulation of of randomness by determinism and of time by space, was proved using different methods by Sipser [Si2] under an *unproven assumption* regarding certain strong expanders.

Consider the following function: On input $<M,x,t>$ the output is a representation of what Turing Machine M does on input x at time t. Where the representation includes the state the machine is in and the location of the heads. Moreover, consider a language L which encodes this function, and let L_n be the restriction of L to strings of length n.

Hypothesis $H1(\varepsilon,n)$: There is a circuit of size $2^{(1-\varepsilon)n}$ that computes L_n.

We will show that if hypothesis H1 is true then some non trivial simulation of time by space is possible, and if it is false then we can use our construction to get a pseudo random bit generator.

Lemma 3.2: If Hypothesis $H1(\varepsilon,n)$ is true for some $\varepsilon>0$ and all sufficiently large n then for some constants $C>1$ and $\varepsilon>0$, and for every function $T(n)=\Omega(C^n)$, $DTIME(T(n))\subset DSPACE(T^{1-\varepsilon}(n))$.

(Compare with [KLi])

Lemma 3.3: If for every $\varepsilon>0$, Hypothesis $H1(\varepsilon,n)$ is false for all sufficiently large n, then for every $\varepsilon>0$ and every $c>0$, there exists a polynomial time generator that converts n^{ε} truly random bits to n bits that look random to any circuit of size n^{c}.

Proof (of lemma 3.2): We will show that (1) if for some $\varepsilon>0$ Hypothesis $H1(\varepsilon,n)$ is true for all n then $L\in DSPACE(2^{(1-\varepsilon)n})$ and that (2) this implies the lemma.

(1) A space-efficient algorithm for L is as follows: The machine tries all circuits of size $2^{(1-\varepsilon)n}$; for each one it checks whether this is indeed the circuit for L. Once it finds the correct circuit, it uses it to look up the answer. Note that checking whether the circuit is the correct one is easy, since it only needs to be consistent between consecutive accesses to the same cell.

(2) Consider any Turing machine M running in $DTIME(T(n))$ where $T(n)=2^{t(n)}$. The result of the Turing machine can be derived by looking whether $<M,x,T(n)>\in L$. This can be done is $DSPACE(2^{(1-\varepsilon)m})$ where m is the size of the input which in this case is $n+t(n)+K$, where K is the length of the description of M. It can be easily checked that the statement of the lemma follows. ■

Note: Actually a stronger statement can be made, as under the assumption H1 the simulation mentioned can even be performed in $\Sigma_2-TIME(T^{(1-\varepsilon)}(n))$.

Proof (of lemma 3.3): First note that if $H1(\varepsilon,n)$ is false then every circuit of size $2^{n/2}$ errs on at least $2^{-\varepsilon n}$ fraction of the inputs, since otherwise there would be at most $2^{(1-\varepsilon)n}$ errors which could be corrected by a table. Next, Yao's Xor lemma (lemma 2.2) allows amplification of the unpredictability by

Xoring disjoint copies of L. By taking $2^{k\varepsilon n}$ disjoint copies (for an appropriately chosen constant k), a function with arbitrary large polynomial hardness can be reached, which can be used as the basis for our generator. ■

The exact statement of the theorem we obtain is thus:

Theorem 9: One of the 2 following possibilities holds:

(1) $BPP \subset \bigcap_{\varepsilon>0} DTIME(2^{n^\varepsilon})$.

(2) There exist $\varepsilon>0$ and $C>1$ such that for any function $T(n)=\Omega(C^n)$, every language in $DTIME(T(n))$ has an algorithm for it that for infinitely many n, runs in SPACE (actually even Σ_2-TIME) $T^{(1-\varepsilon)}(n)$ on all inputs of length n.

Proof: If for every $\varepsilon>0$ Hypothesis $H1(\varepsilon,n)$ holds for only finitely many n then lemma 3.3 assures the existence of pseudorandom generators stretching n^ε bits to n bits, and by lemma 2.1 (1) is true. Otherwise the algorithm in the proof of lemma 3.2 will work for some $\varepsilon>0$ and infinitely many n which implies (2). ■

CHAPTER 3:

Multiparty Protocols and Pseudorandom Generators for Logspace

1. Introduction

This chapter describes how to construct pseudorandom generators for Logspace. The main combinatorial tool we use is a certain multiparty communication game.

Section 2 describes the multiparty communication game and proceeds to give lower bounds on the communication complexity of a certain function. We conclude this section by mentioning some results related to our bounds.

In section 3 we define and discuss pseudorandom generators for Logspace. We then show how to use the lower bounds given in section 2 in order to construct them. We finally give some applications of our pseudorandom generators.

2. Multiparty Communication Complexity

2.1. Definitions

Chandra, Furst and Lipton ([CFL]) introduced the following multiparty communication game: Let $f(x_1, \cdots, x_k)$ be a Boolean function that accepts k arguments each n bits long. There are k parties, each having unlimited computational power, who wish to collaboratively evaluate f. The i'th party knows all the input arguments *except* x_i. They share a blackboard, viewed by all parties, where they can exchange messages. The objective is to minimize the number of bits written on the board.

The game proceeds in rounds. In each round some party writes one bit on the board. The last bit written on the board is considered the outcome of the game and should be the value of $f(x_1, \cdots, x_k)$. The protocol specifies which party does the writing and what is written in each round. It must specify the following information for each possible sequence of bits that is written on the board so far:

(1) Whether the game is over, and in case it is not over, which party writes the next bit: this should be completely determined by the information written on board so far.

(2) What that party writes: this should be a function of the information written on the board so far and of the parts of the input that the party knows.

Definition: The *cost* of a protocol is the number of bits written on the board for the worst case input. The *multiparty communication complexity* of f, $C(f)$, is the minimal cost of a protocol that computes f.

We will also be interested in the number of bits needed in order to compute f correctly on even *most* inputs.

Definition: The *bias* a protocol P achieves on f, $B(P,f)$, is defined to be:

$$B(P,f) = \left|Pr\left[P(x)=f(x)\right]-Pr\left[P(x)\neq f(x)\right]\right|$$

Where $x=(x_1,\cdots,x_k)$ is chosen uniformly over all k-tuples of n-bit strings.

Definition: The *ε-distributional communication complexity* of f, $C_\varepsilon(f)$ is the minimal cost of a protocol which achieves a bias of at least ε on f.

Let us just mention that it is possible to define natural probabilistic or nondeterministic analogues of the multiparty communication complexity. Although we will not be concerned with them, the interested reader may notice that our techniques are strong enough to give lower bounds for all these complexity measures. Also, it is possible to consider the *average* complexities, and our techniques suffice to bound the average complexities as well.

2.2. Previous Work

For the special case $k=2$, this multiparty game is exactly the game standard in communication complexity theory, where one party knows x_1, and the other x_2. This case has been extensively investigated in many different contexts and many different lower bounds appear in the literature ([AUY], [Y1], [BFS], and many more). The distributional communication complexity has been studied as well. Yao [Y3] first considered the distributional communication complexity and proved a lower bound of $\Omega((\log n)^2)$ for the "inner product mod 2" function. Vazirani [Va] improved this bound to $\Omega(n/\log n)$, and Chor and Goldreich [CG] improved it to $\Omega(n)$.

For other values of k less is known. Chandra, Furst and Lipton [CFL] Considered the complexity of the function E_N defined by $E_N(x_1,\cdots,x_k)=1$ iff $x_1+x_2+\cdots+x_k=N$. They showed that for $k=3$ E_N has a

communication complexity of $\Omega(\sqrt{\log N})$ and for general k they only succeeded in showing that the complexity is $\omega(1)$. For the distributional communication complexity, no previous lower bounds were known.

2.3. Cylinder Intersections

In this subsection we study the basic structure that a multiparty protocol induces on the set of possible inputs, the set of k-tuples.

Consider a multiparty protocol for evaluating a function. For every possible k-tuple $x=(x_1, \cdots, x_k)$ a certain communication takes place, and some string is written on the board. The k-tuples may be partitioned according to the string that gets written on the board.

Definition: Let s be a string and P a multiparty protocol. The s-component, $X_{P,s}$, is defined to be the set of k-tuples $x \in (\{0,1\}^n)^k$ such that on input x the protocol P results in exactly s being written on the board.

The s-components have a very special structure, which we will now specify.

Definition: A subset S of k-tuples is called a *cylinder in the i'th dimension*, if membership in S does not depend on the i'th coordinate. A subset of k-tuples is called a *cylinder intersection* if it can be represented as an intersection of cylinders.

Lemma 2.1: For any protocol P and string s, $X_{P,s}$ is a cylinder intersection.

Proof: Define S_i to be the set of k-tuples that is consistent with the communication pattern from the i'th party point of view. I.e.

$$S_i = \left\{ (x_1, \cdots, x_k) : \text{ for } \textit{some } x_i' \ (x_1, \cdots, x_i', \cdots, x_k) \in X_{P,s} \right\}$$

It is clear that for each i, S_i is a cylinder in the i'th coordinate. We will

show that $X_{P,s} = \bigcap_i S_i$.

It is clear that $X_{P,s} \subset \bigcap_i S_i$, it remains to show that $\bigcap_i S_i \subset X_{P,s}$. Let $(x_1, \cdots, x_k) \in \bigcap_i S_i$, then for every i there exists x_i' such that $(x_1, \cdots, x_i', \cdots, x_k) \in X_{P,s}$. We claim that on input $(x_1, \cdots, x_k)$ the protocol will still write s on the board. The reason is that all through the communication process the i'th party cannot distinguish between the input of $(x_1, \cdots, x_k)$ and the input of $(x_1, \cdots, x_i', \cdots, x_k)$. Thus, the bits written on the board will never deviate from s. ■

Given a protocol which computes a function f, the value of f must be constant over any single s-component. Our lower bounds will be based upon the fact that for our particular functions f, any cylinder intersection *must* contain approximately the same number of 1's and 0's of the function.

Definition: Let $f:(\{0,1\}^n)^k \rightarrow \{0,1\}$ be a boolean function. The *discrepancy* of f is

$$\Gamma(f) = \max_S \left| Pr\left[f(x)=1 \text{ and } x \in S\right] - Pr\left[f(x)=0 \text{ and } x \in S\right] \right|$$

where S ranges over all cylinder intersections and x is chosen uniformly over all k-tuples.

Lemma 2.2: For any function f:

$$C(f) \geq \log_2 \left\lfloor \frac{1}{\Gamma(f)} \right\rfloor$$

and

$$C_\epsilon(f) \geq \log_2 \left\lfloor \frac{\epsilon}{\Gamma(f)} \right\rfloor$$

Proof: Consider a protocol P achieving a bias of ε on f. We can compute the bias of P on f as the sum of the biases achieved on the different s-components.

$$Bias(P,f) = \left|Pr\left[P(x)=f(x)\right]-Pr\left[P(x)\neq f(x)\right]\right| \leq$$
$$\sum_s \left|Pr\left[P(x)=f(x) \text{ and } x\in X_{P,s}\right]-Pr\left[P(x)\neq f(x) \text{ and } x\in X_{P,s}\right]\right|$$

where s ranges over all the possible strings that may be written on the board by the protocol.

For any $x\in X_{P,s}$, $P(x)$ was defined to be the last bit of s, thus

$$\left|Pr\left[P(x)=f(x) \text{ and } x\in X_{P,s}\right]-Pr\left[P(x)\neq f(x) \text{ and } x\in X_{P,s}\right]\right| =$$
$$\left|Pr\left[f(x)=1 \text{ and } x\in X_{P,s}\right]-Pr\left[f(x)=0 \text{ and } x\in X_{P,s}\right]\right|$$

Since $X_{P,s}$ is a cylinder intersection, we get that the last quantity is bounded from above by $\Gamma(f)$. Thus, if M is the number of different possible strings that may be written on the board by the protocol P, we get that

$$Bias(P,f) \leq M\cdot\Gamma(f)$$

The statement of the lemma follows since to produce M different strings requires at least $\log_2 M$ bits. ∎

2.4. A lower Bound for Generalized Inner Product

In this subsection we prove a lower bound on the multiparty communication complexity of the generalized inner product.

Definition: The *k-wise generalized inner product* function on n bit strings is defined by

$GIP_{n,k}(x_1,\cdots,x_k)=1$ *iff the number of locations in which all of the*

x_i's have 1 is odd

We will prove a lower bound on the communication complexity of GIP by giving an upper bound to the discrepancy. We first introduce a slightly modified notation to facilitate easier algebraic handling.

Definition: $f(x_1, \cdots, x_k)$ is 1 if $GIP(x_1, \cdots, x_k)=0$ and -1 if $GIP(x_1, \cdots, x_k)=1$.

Definition: Let

$$\Delta^k(n)= \max_{\varphi_1, \cdots, \varphi_k} \left| E_{x_1, \cdots, x_k} f(x_1, \cdots, x_k)\varphi_1(x_1, \cdots, x_k) \cdots \varphi_k(x_1, \cdots, x_k) \right|$$

Where the maximum is taken over all functions $\varphi_i:(\{0,1\}^n)^k \rightarrow \{0,1\}$ s.t. φ_i does not depend on x_i.

The E stands for expected value over all the possible 2^{nk} choices of $x_1, \cdots, x_k$. Note that $\Delta^k(n)$ is exactly $\Gamma(GIP_{k,n})$, the discrepancy of the k-wise inner product function on n-bits.

Lemma 2.3:

$$\Delta^k(n) \leq \mu_k^n$$

where μ_k is given by the recursion: $\mu_1=0$, and $\mu_k = \sqrt{\frac{1+\mu_{k-1}}{2}}$.

Note: It can be shown by induction that $\mu_k \leq 1-4^{1-k}$ which is approximately $e^{-4^{1-k}}$.

Proof: We proceed by induction on k. It is clear that $\Delta^1(n)=0$, expect for the case $n=0$, where we get 1 (let us define $0^0=1$ for this paper).

Let $k \geq 2$. Since φ_k does not depend on x_k,

$$\Delta^k(n) \leq E_{x_1, \cdots, x_{k-1}} \left| E_{x_k} f(x_1, \cdots, x_k)\varphi_1(x_1, \cdots, x_k) \cdots \varphi_{k-1}(x_1, \cdots, x_k) \right|$$

In order to estimate the right-hand side, we will use the following version of

the Cauchy-Schwartz inequality:

Cauchy-Schwartz inequality: For any random variable z: $E[z]^2 \leq E[z^2]$.

Thus our estimate is:

$$\Delta^k(n) \leq \left[\underset{x_1, \cdots, x_{k-1}}{E} \left[\underset{x_k}{E} f(x_1, \cdots, x_k)\varphi_1(x_1, \cdots, x_k) \cdots \varphi_{k-1}(x_1, \cdots, x_k) \right]^2 \right]^{1/2} =$$

$$= \left[\underset{u, v, x_1, \cdots, x_{k-1}}{E} f(x_1, \cdots, x_{k-1}, u) f(x_1, \cdots, x_{k-1}, v) \varphi_1^u \varphi_1^v \cdots \varphi_{k-1}^u \varphi_{k-1}^v \right]^{1/2}$$

where φ_i^u stands for $\varphi_i(x_1, \cdots, x_{k-1}, u)$, and φ_i^v for $\varphi_i(x_1, \cdots, x_{k-1}, v)$.

To estimate this we will need the following observation: For every particular choice of u and v, $f(x_1, \cdots, x_{k-1}, u) f(x_1, \cdots, x_{k-1}, v)$ can be expressed in terms of the function f on $k-1$ strings of a possibly shorter length. Inspection reveals that $f(x_1, \cdots, x_{k-1}, u) f(x_1, \cdots, x_{k-1}, v)$ is simply $f(z_1, \cdots, z_{k-1})$ where z_i is the restriction of x_i to the coordinates j such that $u_j \neq v_j$. We will now view each x_i' as composed of two parts: z_i and y_i, where z_i is the part of the x where $u_j \neq v_j$, and y_i the rest (this is done separately for every u, v).

For every particular choice of u, v and consequently $y_1, \cdots, y_{k-1}$, we define a function of the "z-parts":

$$\xi_i^{u, v, y_1, \cdots, y_{k-1}}(z_1, \cdots, z_{k-1}) = \varphi_i(x_1, \cdots, x_{k-1}, u) \varphi_i(x_1, \cdots, x_{k-1}, v)$$

where the x_i's are obtained by the concatenation of the corresponding y_i and z_i. We can now rewrite the previous estimate as

$$\Delta^k(n) \leq \left[\underset{u, v}{E} \underset{y_1, \cdots, y_{k-1}}{E} S^{u, v, y_1, \cdots, y_{k-1}} \right]^{1/2}$$

where $S^{u, v, y_1, \cdots, y_{k-1}}$ is defined by

$$S^{u, v, y_1, \cdots, y_{k-1}} =$$

$$\underset{z_1,\cdots,z_{k-1}}{E} f(z_1,\cdots,z_{k-1})\xi_1^{u,v,y_1,\cdots,y_{k-1}}(z_1,\cdots,z_{k-1})\cdots\xi_{k-1}^{u,v,y_1,\cdots,y_{k-1}}(z_1,\cdots,z_{k-1})$$

Now, $S^{u,v,y_1,\cdots,y_{k-1}}$ can be estimated via the induction hypothesis. Indeed note that $\xi_i^{u,v,y_1,\cdots,y_{k-1}}$ does not depend on z_i. Thus the previous estimate of $\Delta^k(n)$ is bounded by

$$\Delta^k(n) \le \left[\underset{u,v,y_1,\cdots,y_{k-1}}{E}\Delta^{k-1}(m_{u,v})\right]^{1/2} \le$$

$$\le \left[\underset{u,v,y_1,\cdots,y_{k-1}}{E}\mu_{k-1}^{m_{u,v}}\right]^{1/2}$$

Where $m_{u,v}$ is the length of the strings z_i, which is equal to the number of locations j such that $u_j \neq v_j$.

Since u and v are distributed uniformly in $\{0,1\}^n$, $m_{u,v}$ is distributed according to the binomial distribution. for any constant m, the probability that $m_{u,v}=m$ is exactly $\binom{n}{m}2^{-n}$. Thus the previous estimate is given by:

$$\Delta^k(n) \le \left[\sum_{m=0}^{n}\binom{n}{m}2^{-n}\mu_{k-1}^m\right]^{1/2} =$$

$$=\left[2^{-n}\left[1+\mu_{k-1}\right]^n\right]^{1/2} = \mu_k^n$$

Which completes the proof of the lemma. ■

Combining this bound with lemma 2.4 we get:

Theorem 1:

$$C_\varepsilon\left[GIP_{k,n}\right] = \Omega\left[\frac{n}{4^k}+\log_2\varepsilon\right]$$

■

2.5. Further Results

Our techniques and bounds have several other applications and implications. Since these results are not directly related to pseudorandom generation, we will only describe them briefly here. Complete definitions, discussion and proofs appear in [BNS].

Other lower bounds

Lower bounds on the multiparty communication complexity can also be proved for other functions. In particular, bounds which are slightly stronger than in Theorem 1, are proven for the following function:

Definition: The *quadratic character of the sum mod p* function is:

$QC_{p,k}(x_1, \cdots, x_k)=1$ *iff* $x_1+\cdots+x_k$ *is a quadratic residue mod p*

Theorem: For any n-bit long prime number p:

$$C_\epsilon\left[QC_{p,k}\right] = \Omega\left(\frac{n}{2^k}+\log_2\epsilon\right)$$

Time-Space tradeoffs for Turing machines

Multiparty protocols are a general model of computation, and lower bounds on their complexity may be used to obtain lower bounds in other models. Our lower bounds for the generalized inner product function imply the following time-space tradeoff for general Turing machines:

Theorem: Any k-head Turing machine computing the $k+1$-wise generalized inner product function on n-bit strings requires a time-space tradeoff of $TS=\Omega(n^2)$.

This bound is tight, and significantly better than the previously known tradeoffs for k-head Turing machines ([Ka], [GuS], [DG]).

Branching programs

Chandra Furst and Lipton [CFL] observed how lower bounds for Multiparty protocols can be used to prove length-width tradeoffs for branching programs. Using different techniques, related to those used in [AM], and relying on our lower bounds for multiparty communication complexity, it is possible to obtain length-width tradeoffs for branching programs. These bounds also imply new lower bounds on the size of branching programs and on the size of boolean formula computing certain functions.

3. Pseudorandom Generators for Logspace

In this section we show how to use our lower bounds for multiparty protocols in order to construct pseudorandom generators for Logspace (or generally any small-space machines).

3.1. On Randomized Space

There are quite a few subtle points to consider when defining randomized space bounded classes. We will present here the "correct" definition. For an overview of the subtleties involved refer, e.g., to [].

Definition: A *randomized, space* $s(n)$ Turing machine is defined to have the following properties:

(1) The Turing machine runs in space $s(n)$ on any input of size n.

(2) The Turing machine may flip a fair unbiased coin at any stage.

(3) The Turing machine may never get into an infinite loop, for any sequence of coin flips. In particular, with probability 1 it terminates in time $exp(s(n))$ on any input of length n.

A Turing machine accepts a language L with one-sided error if for every $x \in L$ the machine accepts with probability of at least 1/2 and for any x not in L it rejects with probability 1. A Turing machine accepts a language L with two-sided error if for any $x \in L$, the Turing machine accepts with probability of at least 2/3, and for any x not in L it rejects with probability of at least 2/3.

$RSPACE(s(n))$ is the class of languages accepted with one-sided error by a space $s(n)$ randomized Turing machine. $BPSPACE(s(n))$ is the class of languages accepted with two-sided error by a randomized space $s(n)$ Turing machine. RL is $RSPACE(O(\log n))$, and BPL is $BPSPACE(O(\log n))$.

We want to focus attention on condition (2), the kind of access the machine has to the random bits. As defined, the randomized Turing machine has access to the random bits one by one. When it wants a random bit it can flip a coin, but in no case can it go "back" and review the result of a previous coin flip. Any bit that it wishes to "remember" must be kept in the limited storage. This restriction that the machine does not have multiple, 2-way, access to the random bits is essential, as, for example, random-Logspace machines with 2-way access to the random bits are not even known to be in deterministic polynomial time (in contrast to RL and BPL being in P).

It is interesting to notice that the same apparent difference in power between one-way and two-way access holds also for nondeterministic computation. In the "correct" definition of NL, the Turing machine has one-way access to the nondeterministic bits. If the definition is changed to allow two-way access to nondeterministic bits, then the machines turn out to have the full power of NP.

3.2. Space Bounded Statistical Tests

We are interested in producing pseudorandom sequences that might be used instead of truly random sequences in space bounded computations. Thus the statistical tests that must be passed by the generator are the ones that can be performed by a space bounded Turing machine on its *random source*. Note that this class of tests is a proper subset of the tests that may be performed by space bounded machines on their *input tape*.

We will allow non-uniform statistical tests as well.

Definition: A *space-$s(n)$ statistical test* is a deterministic space $s(n)$ Turing machine M, and an infinite sequence of binary strings $a=(a_1, \cdots, a_n, \ldots)$ called the advice strings. We require that the length of a_n is at most $exp(s(n))$.

The result of the test on input x, $M^a(x)$, is determined as follows: The string a_n, where n is the length of x, is put on a special read-only tape of M called the advice tape. The machine M is run on the advice tape, treating it is a normal input tape. The machine has the following one-way mechanism to access x: at any point it may request the next bit of x (in the same fashion that a randomized Turing machine may request the next random bit).

Notice that these tests have considerable power. We next give some examples of operations that may be performed and combined by Logspace tests:

Consider the input as partitioned into consecutive words, each of some fixed small length ($O(\log n)$). The following questions may all be answered by a Logspace test.

- Count the number of times a certain value appears.
- Compute the average value of a word.
- Compute the standard deviation, or higher moments.
- Compute any of the previous measures for an arbitrary subset of the words, or of the bits.

In fact the vast majority of the statistical tests described by Knuth ([Kn]) lie in this class.

3.3. Definition of Pseudorandom Generators for Space Bounded Computation

A Pseudorandom generator for space $s(n)$ must produce strings that look random to any space $s(n)$ statistical test.

Definition: $G=\{G_n:\{0,1\}^{l(n)}\rightarrow\{0,1\}^n\}$ is called a *pseudorandom generator* for *space* $s(n)$, if for every polynomial $p(n)$, all large enough n, and every space $s(n)$ statistical test M^a,

$$\left|Pr\left[M^a(y)\ accepts\right]-Pr\left[M^a(G(x))\ accepts\right]\right| \leq \frac{1}{p(n)}$$

where y is chosen uniformly in $\{0,1\}^n$, and x uniformly in $\{0,1\}^{l(n)}$.

The first observation we should make when trying to construct a pseudorandom generator for space bounded Turing machines is that without loss of generality we can restrict the class of statistical tests that have to be passed. In a similar fashion to Yao's ([Y2]) results for general (polytime-hard) pseudorandom generators, we show that any generator that passes all space $s(n)$ *prediction* tests will be a pseudorandom generator for space $s(n)$.

Definition: $\{G_n:\{0,1\}^{l(n)}\rightarrow\{0,1\}^n\}$ *passes all space* $s(n)$ *prediction tests* if for every polynomial $p(n)$, every large enough n and every space $s(n)$ statistical

test M^a, and every $1 \le i \le n$,

$$\left| Pr\left[M^a(\textit{first } i-1 \textit{ bits of } G(x)) = i\textit{'th bit of } G(x) \right] - \frac{1}{2} \right| \le \frac{1}{p(n)}$$

Where x is chosen uniformly in $\{0,1\}^{l(n)}$.

Lemma 3.1: G is a pseudorandom generator for $s(n)$ space iff it passes all space $s(n)$ prediction tests.

Proof: Similar to Yao's proof. ■

3.4. Description of the Generator

Our generator is based on a function f that takes k arguments each r bits long, and has high multiparty communication complexity.

The generator

INPUT: the input to the generator will consist of t boolean strings, each r bits long (all together rt random bits).

OUTPUT: each output bit will be of the form $f(S)$ where S is some cardinality k subset of the input strings. The order is extremely important, and we will take *all* the k-subsets in the *colexicographic* order. (I.e. $f(S_1)$ appears before $f(S_2)$ if the latest string in the symmetric difference of S_1 and S_2 is in S_2.) All together there are $\binom{t}{k}$ output bits.

Lemma 3.2: Any Turing machine attempting to predict any bit of the output of the generator from the previous bits with bias ε, requires at least space of $C_\varepsilon(f)/k$.

Proof: Assume not, we will give a multiparty protocol that predicts f with a bias of ε that uses less than $C_\varepsilon(f)$ bits of communication. Suppose $f(S)$ can be predicted by the Turing machine, where $S = \{x_{i_1}, x_{i_2}, \cdots, x_{i_k}\}$, and

$i_1 > i_2 > \cdots > i_k$. We will now show how k parties, the j'th knowing the values of all x_i's in S *except* x_{i_j}, can predict $f(x_{i_1}, x_{i_2}, \cdots, x_{i_k})$, with low communication.

By an averaging argument, it is possible to fix all the other x_i's to constants in some way, while preserving the prediction bias of the Turing machine. Thus we can assume w.l.o.g that all the other x_i's are fixed and known to all the parties beforehand.

The parties will simulate the Turing machine running on the first $i-1$ bits of the output of the generator. The only problem with the simulation is to determine the *input* bits. The first party can simulate the Turing machine by itself until the first time any input involving x_{i_1} comes along. At this point the first party sends the total state of the Turing machine to the second party, who continues with the simulation. Note that this requires only $s(n)$ bits of communication. The second party can now continue the simulation until x_{i_2} comes along. And now the second party sends the state of the machine to the third party which now continues the simulation.

It is important to notice that, because of the ordering of the subsets we chose, by the time the k'th player begins the simulation, he can continue it all the way until the prediction of the i'th bit, $f(S)$. The total number of bits communicated is $(k-1)s(n)$. ■

We can now state the main theorem we reach in this section:

Theorem 2: For some constants $c_1, c_2 > 0$, there exists (an explicit) pseudorandom generator, $G = \{G_n : \{0,1\}^{l(n)} \rightarrow \{0,1\}^n\}$, for space $2^{c_1\sqrt{\log n}}$, where $l(n) = 2^{c_2\sqrt{\log n}}$. Moreover, G can be computed by a Logspace Turing machine (having multiple access to its input bits).

Proof: We use the construction of the generator described in this section, with f being the "generalized inner product" function, $k=O(\sqrt{\log n})$, $t=2k$, and $r=exp(O(\sqrt{\log n}))$. Theorem 1 guarantees the high communication complexity of this function, and thus lemma 3.2 shows that the generator passes all prediction tests, and lemma 3.1 concludes that it is a pseudorandom generator. ■

3.5. One way vs. Two-way Access to Randomness

Our generator sheds some light on the difference between one-way and two-way access to the random bits given to Logspace machines. We show that 2-way access is better, at least in the sense that fewer random bits are necessary.

Corollary 3.3: A randomized Logspace Turing machine with one-way access to the random bits that uses a polynomial number of random bits may be simulated by a randomized Logspace machine with 2-way access to the random bits that uses only $2^{O\sqrt{\log n}}$ random bits.

Proof: Our generator can be implemented by a Logspace machine having 2-way access to the random bits. The generator can be easily run "on the fly", and generate one bit at a time to supply to the original, simulated machine, whenever it flips a coin. ■

3.6. Universal Sequences

One of the most interesting subclasses of Logspace statistical tests are those related to walks on graphs. Given a graph H in the advice tape, a Logspace machine can treat the input to the test as directions for a walk on the graph and perform the walk. Thus the output of a pseudorandom generator for Logspace will behave like a random walk on any graph. We use

this fact in order to construct universal traversal sequences.

Definition: A graph is called (d,n)-*labeled* if it is a d-regular graph on n vertices and the edges adjacent to each vertex are labeled by a permutation of $\{1,\cdots,d\}$ (an edge may be labeled differently at each of its 2 vertices). A string $w\varepsilon\{1,\cdots,d\}^*$ is said to *cover* a (n,d)-labeled graph, if w, when treated as directions to a walk on the graph, visits all vertices of the graph, whatever the starting vertex is.

Definition: A string $w\varepsilon\{1,\cdots,d\}^*$ is said to be a (n,d) *universal traversal sequence* if it covers every (d,n)-labeled graph.

[AKLLS] first showed that a random string of length $O(d^2n^3\log n)$ is a (d,n)-universal sequence with high probability. Results shown in [KLNS] imply that a random string of length $O(dn^3\log n)$ actually suffices. However, explicit construction of short universal sequences is more difficult. Explicit constructions are known for two special cases: for $d=2$ an explicit construction is known of polynomial length universal sequences ([Is]); and for $d=n$ an explicit construction of length $n^{\log n}$ is known ([KPS]). Our pseudorandom sequences allow us to give (d,n)-universal sequences of length $2^{2^{O(\sqrt{\log n})}}$ for *all* values of d.

We will be using the output of the generator as a random walk. A technical issue that should be mentioned is that we need to convert the *binary* string which is the output of the generator to a string in $\{1,\cdots,d\}$. One way to do this is to take every $5\log n$ bits consecutive bits modulo d. This way a uniform distribution on binary string will be converted to an almost uniform distribution on walks.

Lemma 3.4: Let $G=\{G_n:\{0,1\}^{l(n)}\rightarrow\{0,1\}^{n^3}\}$ be a pseudorandom generator for Logspace. Then for every (d,n)-labeled graph H, $G(x)$ (converted as mentioned) will cover H with probability at least 1/2 (probability taken over a

random choice of x).

Proof: The description of H can be put in the oracle, and then a Logspace machine can perform the walk on H given by its input. A truly random string y of length n^4 converted in this manner will result in a nearly uniformly random walk on H, and as such will visit every vertex, starting from every vertex with probability of at least $1-1/3n^2$. A Logspace machine can determine whether vertex i is reached in a walk starting from vertex j, and thus for every i,j, the probability that $G(x)$ (converted to a walk) will reach vertex i starting from vertex j, should be at least $1-1/2n^2$. Thus the probability that there exist i,j such that the walk from i does not reach j is at most 1/2. ■

Lemma 3.5: Let $G=\{G_n:\{0,1\}^{l(n)}\rightarrow\{0,1\}^{n^4}\}$ be a pseudorandom generator for Logspace. Then the string achieved by the concatenation of $G(x)$ for all possible $2^{l(n)}$ values of x (and converted as mentioned) is a (d,n) universal traversal sequence.

Proof: For each (n,d)-labeled graph H, half the substrings of the form $G(x)$ will cover H. Thus when one of these substrings is reached, whatever vertex the walk is in, H will be covered by it. ■

Applying our generator to this lemma we get:

Theorem 3: For every d and n, there exists an (explicitly given) (d,n) universal traversal sequence of length $2^{2^{O(\sqrt{\log n})}}$. Moreover the sequence can be constructed by a Turing machine running in space logarithmic in the length of the sequence. ■

References

[AH] L.M. Adleman and M.A. Huang, "Recognizing primes in random polynomial time", STOC '87.

[AKLLR] R. Aleliunas, R.M. Karp, R.J. Lipton, L. Lovasz and C. Rackoff, "Random walks, universal traversing sequences and the complexity of maze problems", FOCS '79.

[AM] N. Alon and W. Maass, "Meanders, Ramsey theory and lower bounds for branching programs", FOCS '86.

[AUY] A.V. Aho, J.D. Ullman and M. Yanakakis, "On notions of information transfer in VLSI circuits", STOC'83.

[AW] M. Ajtai and A. Wigderson, "Deterministic simulation of Probabilistic constant depth circuits", 26th FOCS, pp. 11-19, 1985.

[Ba] L. Babai, "Trading group theory for randomness", 17th STOC, pp. 421-429, 1975.

[BCDRT] A. Borodin, S.A. Cook, P.W. Dymond, W.L. Ruzzo and M. Tompa, "Two applications of inductive counting for complementation problems", manuscript, 1988.

[BFS] L. Babai, P. Frankl and J. Simon, "Complexity classes in communication complexity theory", FOCS '86.

[BG] C.H. Bennett and J. Gill, "Relative to a random oracle A, $P^A \neq NP^A \neq Co-NP^A$ with probability 1", SIAM J. Comp. 10, 1981.

[BaM] L. Babai and S. Moran, "Arthur Merlin games: a randomized proof system, and a hierarchy of complexity classes", Manuscript, 1988.

[BM] M. Blum and S. Micali, "How to generate cryptographically strong sequences of pseudo random bits", 23rd FOCS, pp. 112-117, 1982.

[BNS] L. Babai, N. Nisan and M. Szegedy, "Multiparty protocols and logspace hard pseudorandom sequences", manuscript, 1988.

[CFL] A. Chandra, M. Furst and R.J. Lipton, "Multiparty protocols", FOCS '83.

[CG] B. Chor and O. Goldreich, "Unbiased bits from weak sources of randomness", FOCS '85.

[CKS] A. Chandra, D. Kozen and L. Stockmeyer, "Alternation", J. ACM 28, 1981.

[DG] P. Duris and Z. Galil, "A time-space tradeoff for language recognition", Math. Systems theory 17 3-12, 1984.

[FLS] M. Furst, R.J. Lipton and L. Stockmeyer, "Pseudo random number generation and space complexity", Information and Control, Vol. 64, 1985.

[GM] S. Goldwasser and S. Micali, "Probabilistic Encryption", JCSS Vol. 28, No. 2, 1984.

[GMW] O. Goldreich, S. Micali and A. Wigderson, "Proofs that yield nothing but their validity and a methodology of cryptographic protocol design", FOCS '86.

[GS] S. Goldwasser and M. Sipser, "Private coins vs. public coins in interactive proof systems", 18th STOC, pp. 59-68, 1986.

[GuS] Y. Gurevich and S. Shelah, "Nondeterministic linear tasks may require substantially nonlinear deterministic time in the case of

sublinear work space", STOC '88.

[Ha] J. Hastad, "Computational limitations for small depth circuits", Ph.D. thesis, M.I.T. press, 1986.

[HPV] J. Hopcroft, W. Paul and L. Valiant, "On time versus space and related problems", 16th FOCS, 1975.

[ILL] R. Impagliazzo L. Levin and M. Luby, "Pseudorandom generators from any one-way function", manuscript, 1988.

[Is] S. Istrail, "Polynomial universal traversing sequences for cycles are constructible", STOC '88.

[Ka] M. Karchmer, "Two time-space tradeoffs for element distinctness", *Theoretical Computer Science* 47, 1986.

[KLi] R. M. Karp and R. Lipton, "Turing machines that take advice", Enseign. Math. 28, pp. 191-209, 1982.

[KLNS] J.D. Kahn, N. Linial, N. Nisan and M.E. Saks, "On the cover time of random walks in graphs", *Journal of Theoretical Probability*, Vol. 2, No. 1, 1988.

[KLu] R.M. Karp and M. Luby, "Monte-Carlo algorithms for enumeration and reliability problems", 24th FOCS, pp. 56-64, 1983.

[Kn] D. Knuth, "The art of computer programming, Vol II".

[Ku] S. A. Kurtz, "A note on randomized polynomial time, SIAM J. Comp., Vol. 16, No. 5, 1987.

[KUW] R.M. Karp, E. Upfal and A. Wigderson, "Constructing a maximum matching is in random NC", *Combinatorica* 6(1), pp. 35-48, 1986.

[KPS] H. Karlof, R. Paturi and J. Simon, "Universal sequences of length $n^{O(\log n)}$ for cliques", manuscript, 1987.

[LR] M. Luby and C. Rackoff, "Pseudorandom permutation generators and cryptographic composition", STOC '86.

[MVV] K. Mulmuley, U.V. Vazirani and V.V. Vazirani, "Matching is as easy as matrix inversion", STOC '87.

[NW1] N. Nisan and A. Wigderson. "Pseudo random bits for constant depth circuits", manuscript, 1988.

[NW2] N. Nisan and A. Wigderson, "Hardness vs. Randomness", FOCS '88.

[RT] J.H. Reif and J.D. Tygar, "Towards a theory of parallel randomized computation", TR-07-84, Aiken computation lab., Harvard university, 1984.

[Si1] M. Sipser, "A complexity theoretic approach to randomness", 15th STOC, 330-335, 1983.

[Si2] M. Sipser, "Expanders, Randomness, or Time vs. Space", Structure in Complexity Theory, Lecture notes in Computer Science, No. 223, Ed.

[Sh] A. Shamir, "On the generation of cryptographically strong pseudo-random sequences", 8th ICALP, Lecture notes in Comp. Sci. 62, Springer-Verleg, pp. 544-550, 1981.

[SS] R. Solovay and V. Strassen, "A fast Monte-carlo test for primality", *Siam J. of Computing* 6, pp. 84-85, 1977.

[St] L. Stockmeyer, "The polynomial time hierarchy", Theor. Comp. Sci. 3, No. 1, 1976.

[Va] U.V. Vazirani, "Towards a strong communication complexity theory or generatoing quasi-random sequences from two communicating semirandom sources", STOC '85.

[Y1] A.C. Yao, "Some complexity questions related to distributed computing", STOC '79.

[Y2] A.C. Yao, "Theory and applications of trapdoor functions", 23rd FOCS, pp. 80-91, 1982.

[Y3] A.C. Yao, "Lower bounds by probabilistic arguments", STOC '83.