Hardness vs. Randomness

Noam Nisan [1]

Institute of Computer Science
Hebrew University of Jerusalem, Israel

Avi Wigderson [2]

Institute of Computer Science
Hebrew University of Jerusalem, Israel

ABSTRACT

We present a simple new construction of a pseudorandom bit generator, based on the constant depth generators of [N]. It stretches 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 problem of proving lower bounds and the problem of generating good pseudorandom sequences.

Our 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 an unconditionally proven pseudorandom generator for constant depth circuits. As an application of this generator we characterize the power of NP with a random oracle.

1. Introduction

The fundamental idea of trading hardness for randomness is due to Shamir [Sh], who suggested that the RSA function can be used to construct good pseudo-random sequences. The first secure pseudo-random bit-generator was built by Blum and Micali

---

1 This work was done while the first author was a student in the University of California at Berkeley.

2 Supported by Israel National Academy of Science grant No. 328071, by the Alon Fellowship, and by NSF grant CCR8612563.

[BlM], who used the intractability of the Discrete Logarithm function. These ideas were generalized by Yao [Ya], who showed that any one-way permutation can be used to construct generators that fool every polynomial time computation. This result gave the first explicit hardness-randomness trade-off: if no poly-size circuit can invert the one-way permutation, then $RP \subset \bigcap_{\varepsilon>0} DTIME(2^{n^{\varepsilon}})$. Yao's result was recently generalized by Impagliazzo, Levin and Luby [ILL] who succeeded in constructing a pseudorandom generator based on an arbitrary one-way function.

In all these papers, the generator uses the one-way function f essentially as follows: >From a random string X_0 (the seed), it computes a sequence $\{X_i\}$ by $X_{i+1}=f(X_i)$. The output bits b_i depend on this sequence. The heart of the argument is then showing that a small circuit that is not fooled by the bit sequence $\{b_i\}$ can be used to compute f^{-1}, contradicting its assumed hardness. A deterministic simulation of a randomized algorithm then proceeds by trying all possible seeds X_0.

These pseudorandom generators can all be computed by polynomial time Turing machines, and in fact this requirement is usually considered part of the definition of pseudorandom generators. While this requirement is needed for various cryptographic applications, it is not needed for "simulation of randomized algorithms" purposes, which are the focus of interest in this paper. We thus propose to separate the requirements regarding the running time of the generator form the requirements regarding the "pseudorandomness" (security) of its output. In this paper we will thus use the term "pseudorandom generator" for any function whose output "looks random" (to some class of algorithms). Of course, for our pseudorandom generators to be of any interest we will separately need to show that they can be computed "sufficiently fast".

All of the pseudorandom generators mentioned above share the following limitations:

(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. The only known parallel pseudorandom generator [RT] is based on a very specific function. There is no known construction of pseudorandom generators for NC that is based on a general complexity assumption about NC.

We propose here a new construction of a generator, based on the constant depth generators of [N], which avoids both problems. This generator does not run in polynomial time, but it can be computed sufficiently fast for our simulation purposes. The generator can be based on the hardness of approximation of an arbitrary function in *EXPTIME*, and is completely parallel and thus can be applied to other complexity classes. (To "approximate a function" means to agree with it on a large fraction of inputs -- exact definitions appear in section 2.1). Simply, each output bit b_i of the generator is computed by applying f to a subset S_i of the bits in the seed X_0. These subsets (up to exponentially many) are explicitly described and have the property that every pair of them is nearly disjoint. Here this property is the heart of ensuring the quality of the bits.

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".

Theorem 1: For every function s, $m \le s(m) \le 2^m$ the following are equivalent:

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

(2) For some $c > 0$ there exists a Pseudo-random generator $\{G_m : \{0,1\}^m \rightarrow \{0,1\}^{s(m^c)}\}$ that runs in time exponential in m and its output looks random to any circuit of size $s(m^c)$.

This theorem 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 weaker assumptions but yields less efficient pseudorandom generators. This loss, however, does not have any effect when using pseudorandom generators for the deterministic simulation of randomized algorithms.

This construction has many implications and we describe some of them in section three.

Hardness - Randomness tradeoff

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; a strong enough assumption implies P=BPP. An example of this tradeoff is:

Theorem 2: If EXPTIME cannot be approximated by polynomial size circuits then $RP \subset \bigcap_{\varepsilon > 0} DTIME(2^{n^{\varepsilon}})$.

Comment: All our simulation results are stated for one-sided error classes, but obviously hold for the two-sided analogs as well.

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.

In [Ya], the same consequence is obtained assuming one-way permutations exist, an assumption which is stronger than $NP \cap Co-NP$ cannot be approximated by polynomial size circuits.

Randomized Parallel Computation

Reif and Tygar [RT] considered simulation of probabilistic parallel algorithms under intractability assumptions. They showed how to parallelize the Blum-Micali type generator over a particular function (inverse mod p), and thus get a pseudorandom generator for *NC* assuming that this function (which is in *P*) cannot be approximated by NC circuits. Under this assumption they obtain $RNC \subset \bigcap_{\varepsilon>0} DSPACE(n^{\varepsilon})$. As our construction is parallel, we can obtain:

Theorem 3: If PSPACE cannot be approximated by NC circuits then $RNC \subset \bigcap_{\varepsilon>0} DSPACE(n^{\varepsilon})$

Randomized Constant Depth Circuits

Ajtai and Wigderson [AW] studied the simulation of probabilistic constant- depth circuits, since for them lower bounds exist. They devised a complicated generator, based on the proof methods of the parity lower bound, that gave the first nontrivial simulation result proven without any assumptions: $RAC^0 \subset \bigcap_{\varepsilon>0} DSPACE(n^{\varepsilon})$. Our construction enables us to use directly the best parity lower bound [Ha] and obtain the stronger result:

Theorem 5: $RAC^0 \subset \bigcup_{c} DSPACE((\log n)^c)$

This result is *not* based on any unproven assumptions. The only other complexity class for which pseudorandom generators are unconditionally proven to exist is Logspace [BNS].

Random Oracles

The power of random oracles is an old subject of interest [BG, BGS]. For a complexity class *C*, define $almost\!-\!C = \{L : L \in C^A$ for almost all oracles $A\}$.

Baker and Gill [BG] proved that $almost\!-\!P = BPP$, suggesting that *BPP* is the right probabilistic analog of *P*. Babai [Ba] introduced the class *AM* (Arthur-Merlin games) and proposed it as a probabilistic analog of *NP*. We justify this intuition, answering an open question of Babai and Sipser (see [BM]).

Theorem 6: $almost\!-\!NP = AM$.

The proof relies on a description of $almost\!-\!NP$ as a probabilistic, exponential size, constant-depth circuit, and our generator. A similar consideration, together with Sipser's result that $BPP \subset PH$ [Si1], implies the surprising fact that random oracles do not help the polynomial time hierarchy.

Theorem 7: $almost\!-\!PH = PH$.

BPP and the Polynomial Time Hierarchy

In [Si1] Sipser showed that *BPP* is contained in the polynomial time hierarchy. Gacs improved on this and showed that *BPP* is actually contained in $\Sigma_2 \cap \Pi_2$. Using our generator, we give a completely different, simple proof of this fact.

Time vs. Space and Randomness vs. Determinism

In [Si2] Sipser made the striking observation that efficient deterministic simulation of probabilistic algorithms is intimately related to efficient simulation of time by space (in a certain weak sense).

Assuming that it is possible to explicitly construct certain strong expanders, he proved that either $RP = P$, or else some nontrivial space-efficient simulation of time-bounded Turing machines is possible. (A space simulation which is significantly better than the best unconditional bound of $t(n) / \log t(n)$ of Hopcroft, Paul and Valiant [HPV]).

We use our generator to give a completely different proof of a slightly weaker relation, but using no unproven assumption.

Note: Since this manuscript was originally written [BFNW] have strengthened some of the results appearing here.

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} (for some $0 < \varepsilon < 1$); "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 [Ya] 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 constructible 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 [Ya] shows how the closeness of approximation can be amplified by xor-ing multiple copies of f. A full proof of this lemma may be found in [BH].

Lemma 2.2 (Yao) : Let $f_1, \cdots, f_k$ all be (ε,S)-hard. Then for any $\delta>0$, the function $f(x_1 \cdots x_k)$ defined by

$$f(x_1 \cdots x_k) = \sum_{i=1}^{k} f_i(x_i) \ (mod\ 2)$$

is $(\varepsilon^k+\delta, \delta^2(1-\varepsilon)^2 S)$-hard.

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 such that $m \le s(m) \le 2^m$; 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 $H_{f'}(m) \ge s(m^c)$.

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 $i \ne j$:

$$\left| S_i \cap S_j \right| \le k$$

A $n \times l$ 0-1 matrix is called a (k,m)-design if the collection of its n rows, interpreted as subsets of $\{1..l\}$, is 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 $H_f(m) \ge n^2$; 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,

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

where x is chosen uniformly in $\{0,1\}^l$, and y is chosen uniformly in $\{0,1\}^n$. We first show, as in [GM] and in [Ya], 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 [Ya] 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 into 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, we can rewrite

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

where x is chosen at random, as the average over all possible choices of $x_{m+1} \cdots x_l$ of the same expression where only $x_1 \cdots x_m$ are chosen at random. It follows that for some particular choice of values $c_{m+1} \cdots c_l$ for $x_{m+1} \cdots x_l$, setting $x_j = c_j$ for all $m < j \le l$ 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 $G : l \to n$. 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 $(logn,m)$-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 all 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 $l = O(m^2)$. Moreover, the matrix can be computed by a Turing machine running in space $O(logn)$.

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 all integers n and m, where $m = C \log n$, there exists a $n \times l$ matrix which is a $(\log n, m)$-design where $l = O(C^2 \log n)$. 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. (A random subset of size size m is expected to intersect a single given set of size m in m^2/l points; the probability that the intersection is more than, say, twice as large can be bounded by $1/n$ using Chernoff bounds.) 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 s, $l \le s(l) \le 2^l$ 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* $G : l \to l+1$.

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

Note: We assume here that $(s(l))^c \le s(l^c)$ (as is true for most functions of interest as size bounds), otherwise the expression $s(l^c)$ should be changed everywhere to $s(l^c)^c$.

Proof:

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

(4) -> (3) is trivial.

(3) -> (1) [1] 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 compute it since that circuit would distinguish between the output of G and between truly random strings. If G happens to be 1-1 then it is also clear that no circuit of size $s(l^c)$ can even approximate this language. However, if G is not 1-1 then in order to obtain a function which cannot even be approximated we will require the following fact which developed from the work on random-self-reducability: If every function in EXPTIME can be approximated to within $1/n^2$ by circuits of size $s(n)$ then every function in EXPTIME can be computed exactly by circuits of size $s(n) poly(n)$. The proof of this fact proceeds by taking the multi-linear extension of the function, and using the radnom-self-reducability of the extension to correct errors. See [BFNW] for details and references.

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 $G : l \to n$, 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

1 Our original paper contained an error in the proof of this implication (which is the converse of the main result.) The error was pointed to us by Oded Goldreich. The correction appearing here uses an idea from [BFNW].

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 $G: n^{\varepsilon} \rightarrow n$.

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 $2^{n^{\varepsilon}}$ for some $\varepsilon > 0$. Or,

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

Then

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

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

(3) $BPP = P$.

respectively.

Note: Here we mean that for $i = 1, 2, 3$, assumption i implies conclusion i.

Proof: using theorem 1, (1) implies the existence of a quick pseudorandom generator $G: n^{\varepsilon} \rightarrow n$ for every $\varepsilon > 0$, 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 $G: C \log n \rightarrow n$ 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 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.

Note: Here we mean that for $i=1,2$, assumption i implies conclusion i.

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. These results appear also in a previous paper of ours [N] with more complete proofs and some extensions.

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:

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

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 that with high probability outputs a number which is within a factor of 2 of the the number of satisfying assignments. It is not difficult to see that this algorithm can be implemented by a random-AC^0 circuit of depth 4. The output of our generator may be used by this circuit instead of truly random bits without significantly changing the probability that the output of the circuit is in any fixed range. (Since otherwise we could add a comparator to the circuit and obtain a 1-bit constant depth circuit that distinguished between random and pseudorandom strings.) □

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:

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

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 the fact that $BPP \subset \Sigma_2 \cap \Pi_2$ (see next subsection) 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 $\Sigma_2 \cap \Pi_2$. In this section we give a new simple proof of this fact.

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

Proof: Because BPP is closed under complement 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 F: 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 F, 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 $T(n)=\Omega(C^n)$, $DTIME(T(n)) \subset DSPACE(T^{1-\varepsilon'}(n))$.

(This result is similar to results in [KLi] "translating" non-uniform upper bounds to uniform ones.)

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_n. 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 $T(n)=2^{t(n)}$. Whether the Turing machine accepts or not can be derived from the value of $F(<M,x,T(n)>)$ which is encoded by L_m, where m is the size of the input which in this case is $n+t(n)+K$, where K (a constant) is the length of the description of M. This can be computed in $DSPACE(2^{(1-\varepsilon)m})$. For proper choices of C and ε', $(1-\varepsilon)m\leq(1-\varepsilon')t(n)$, and 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: for any constant c', (assuming ε is small enough,) there exists a constant d so that by taking $2^{d\varepsilon n}$ disjoint copies, we get a function over $N=n2^{d\varepsilon n}$ variables such that every circuit of size, say, $2^{n/4}$ cannot achieve bias of better than $N^{-c'}$. Thus this new fuction has hardness $H(N)\geq N^{c'}$. This hardness suffices (by theorem 1) for a constructing a pseudorandom generator as required by the lemma. □

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). □

4. Acknowledgements

We would like to thank Laszlo Babai for suggesting AM=almost-NP? as an application of our result. We thank Silvio Micali for allowing us to steal the title of his Ph.D. thesis. We thank Oded Goldreich for pointing out an error in a previous version of this paper.

5. References

[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.

[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", JCSS 36(2), pp. 254-276, 1988.

[BFNW] L. Babai, L. Fortnow, N. Nisan and A. Wigderson, "BPP has weak subexponential simulations unless EXPTIME has publishable proofs", proceedings of structures in complexity theory, 1991.

[BH] R. Boppana and R. Hirschfeld, "Pseudorandom generators and complexity classes", In "Randomness and Computation", volume 5, Editor S. Micali, of Advances in Computing Research, JAI Press, Greenwich, 1989, 1--26.

[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", 21st STOC, pp. 1-11, 1989.

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

[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.

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

[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", 21st STOC, 1989.

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

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

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

[N] N. Nisan. "Pseudo random bits for constant depth circuits", Combinatorica, 11(1), pp. 63-70, 1991.

[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. G. Goos, J. Hartmanis, pp. 325-329.

[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.

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

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