Typically-correct derandomization*

Ronen Shaltiel†

June 22, 2010

Abstract

A fundamental question in complexity theory is whether every randomized polynomial time algorithm can be simulated by a deterministic polynomial time algorithm (that is, whether BPP=P). A beautiful theory of derandomization was developed in recent years in attempt to solve this problem.

In this article we survey some recent work on relaxed notions of derandomization that allow the deterministic simulation to err on some inputs. We use this opportunity to also provide a brief overview to some results and research directions in "classical derandomization".

1 Introduction

Randomized algorithms are algorithms that get an additional input which is a sequence of independent coin tosses and are allowed to err with small probability. Randomized algorithms play an important role in many areas in computer science. For some choices of computational resources it is known that randomized algorithms can be significantly more efficient than deterministic ones. A fundamental longstanding open problem asks whether BPP=P (namely, can every randomized polynomial-time algorithm be simulated by a polynomial-time deterministic algorithm). A long line of research is devoted to studying this problem (see, e.g., [Mil01, Kab02, Imp06] for survey articles). This research led to a beautiful theory of derandomization which attempts to provide efficient deterministic simulation of randomized algorithms in various settings. Some highlights of this research are "hardness versus randomness tradeoffs" showing that BPP=P assuming certain circuit lower bounds [IW97] (see also [BM84, Yao82, NW94, BFNW93, STV01, SU05, Uma03]), and that the statement BPP=P entails certain circuit lower bounds that seem hard to prove using current techniques [KI04, IKW02].

1.1 Relaxed notions of derandomization

In this article we survey some recent research which studies relaxed notions of derandomization that allow the deterministic simulation to err on some inputs. We will refer to the standard approach (which requires the simulation to succeed on all inputs) as "always-correct derandomization". Several relaxations of this notion were considered in the literature. The most simple approach is a quantitative approach suggested in [GW02]. This notion allows the deterministic simulation to err

*This survey article appeared as SIGACT News complexity theory column 66 with an introduction by Lane Hemaspaandra. See SIGACT News 41(2):57-72, 2010.

†University of Haifa, Mount Carmel 31905, Israel. ronen@cs.haifa.ac.il. Supported by BSF grant 2004329 and ISF grant 686/07.

on a small number of inputs of every input length. We refer to this approach as "typically-correct derandomization".

Typically-correct derandomization can be viewed as measuring the success probability of the deterministic simulation when the input is chosen at random from the uniform distribution. A generalization is to consider some subset of probability distributions over inputs and require that the deterministic simulation succeeds with high probability for every distribution in the subset. This approach was suggested by [IW01] for the subset of distributions that are samplable in polynomial time. This notion is closely related to Levin's theory of average-case complexity [Lev86]. If one accepts the assumption that "real life inputs" are chosen from efficiently samplable distributions then such a deterministic simulation is guaranteed to do well in "real life scenarios".

We can also view this setup as a game between an *algorithm designer* who designs the deterministic simulation and a *refuter* who tries to produce inputs on which the simulation fails. The deterministic simulation is considered successful if it is infeasible for the refuter to produce inputs on which the simulation fails. The notion of [IW01] considers refuters that are polynomial time randomized algorithms. A generalization by [Kab01] considers refuters from various complexity classes. We refer to this approach as "derandomization on feasibly generated inputs". The reader is referred to [Kab02] for a survey article that discusses this line of work (which is often referred to as "uniform derandomization").

1.2 Organization of this article

We assume that the reader has basic knowledge in computational complexity and start by recalling the definition of randomized algorithms in Section 1.3. In Section 2 we give a brief overview to the area of standard (that is always-correct) derandomization. In Section 3 we discuss derandomization on feasibly generated inputs. We do not go into details and our goal is mainly to explain the relationship between this notion and always-correct derandomization. In Section 4 we describe some recent results on typically-correct derandomization. Our aim is to try and highlight some of the ideas and techniques that were developed in this framework.

1.3 Notation for randomized algorithms

We now give formal definition of randomized algorithms that will be used throughout the paper. A randomized procedure is a function $A : \{0,1\}^n \times \{0,1\}^m \to \{0,1\}$. We refer to n as the *input length* and to m as the *number of coins* of A. We say that procedure A *computes* a function $f : \{0,1\}^n \to \{0,1\}$ with success $1-\rho > 1/2$ if for every input $x \in \{0,1\}^n$

$$\Pr_{R \leftarrow U_m}[A(x,R) = f(x)] \geq 1-\rho.$$

where U_m denotes the uniform distribution over strings of length m and the notation $R \leftarrow U_m$ denotes the experiment in which R is chosen at random according to the distribution U_m. The definition says that for every input x, the probability that A errs on x is bounded by ρ and we refer to ρ as the *error* of A. If we omit ρ and say that A computes f then we mean that A computes f with success $2/3$. We use the word "procedure" above to emphasize that we consider a fixed input length n and do not bound the complexity of A.

A randomized algorithm receives inputs of varying lengths and is associated with a function $m : \mathbb{N} \to \mathbb{N}$ measuring the number of coins used by A on inputs of length n. When receiving an input $x \in \{0,1\}^n$, algorithm A also expects to receive a second input $r \in \{0,1\}^{m(n)}$. We say that

A computes a Boolean function f with success $1-\rho$ (where $\rho : \mathbb{N} \to [0,1]$) if for every input length n, A computes f with success $1-\rho(n)$ when restricted to that input length.

A randomized polynomial time algorithms is a randomized algorithms that uses $m(n) = \text{poly}(n)$ coins and is computable by a polynomial time Turing machine. The class of all functions computed by polynomial time randomized algorithm is called BPP. We can also consider other classes of randomized algorithms by modifying the notion of efficiency. For example, a randomized algorithm is in randomized AC^0 if it uses a polynomial number of coins and can be computed in uniform-AC^0. That is, there is a uniform family of polynomial size constant depth circuits $C = \{C_n\}$ which implements $A(x,r)$.

2 Brief overview on always-correct derandomization

The research on relaxed notions of derandomization builds on, and should be compared to, the research on always-correct derandomization. In this section we give a brief overview to the area of derandomization. This overview is not meant to be comprehensive and focuses on issues that will be relevant later on.

2.1 Pseudorandom generators

The key concept in derandomization is that of a pseudorandom generator (PRG). This is a procedure $G : \{0,1\}^d \to \{0,1\}^m$ that stretches a short string called "the seed" into a much longer string. Pseudorandomness means that the distribution $G(U_d)$ is indistinguishable from U_m. Making this definition precise requires specifying the class of procedures trying to distinguish the two distributions. We say that G is ϵ-pseudorandom for a class $\mathcal{C}$ of functions $C : \{0,1\}^m \to \{0,1\}$ if for every function C in the class $\mathcal{C}$,

$$|\Pr_{S \leftarrow U_d}[C(G(S)) = 1] - \Pr_{R \leftarrow U_m}[C(R) = 1]| \leq \epsilon.$$

The value of ϵ is fixed to $1/10$ if it is not specified and we sometimes loosely say that G *fools* $\mathcal{C}$. The definition above considers PRGs with fixed seed length and output length. We typically consider PRGs with varying seed length and output length and the key parameters are:

- The *stretch* which is the relationship between seed length d and output length m.
- The class $\mathcal{C}$ that is fooled by the PRG on output length m.
- The running time of the PRG which is typically measured as a function of the seed length d.

Pseudorandom generators can be used to reduce the number of random coins used by randomized procedures. More precisely, given a randomized procedure $A : \{0,1\}^n \times \{0,1\}^m \to \{0,1\}$ that computes some function f with success $1-\rho$ and a pseudorandom generator $G : \{0,1\}^d \to \{0,1\}^m$ that is ϵ-pseudorandom for the class $\mathcal{C} = \{A(x,\cdot) : x \in \{0,1\}^n\}$ we have that the randomized procedure $A' : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^m$ defined by $A'(x,s) = A(x,G(s))$ computes f with success $1-\rho-\epsilon$ (which is greater than half for our standard choices of ρ, ϵ). Indeed, if A' does not agree with A on some input x then the function $C(r) = A(x,r)$ distinguishes between $G(U_d)$ and U_m.

Every randomized procedure A' that uses d coins can be simulated by a deterministic procedure B defined as follows: On input x, run $A'(x,s)$ for all $s \in \{0,1\}^d$ and output the most likely outcome. The use of a PRG to reduces the number of coins used by deterministic algorithms, making this deterministic simulation more efficient.

2.2 Using pseudorandom generators to derandomize BPP

The simulation above gives a general recipe for derandomization of every randomized algorithm. In particular "wholesale derandomization" of all polynomial time randomized algorithms can be achieved if we can construct the appropriate PRG. In the paragraph below we discuss requirements from such a PRG. (Jumping ahead we mention that we do not know how to construct such PRGs unconditionally and all current constructions rely on unproven assumptions).

A randomized algorithm that runs in time $t(n) = \text{poly}(n)$ uses at most $t(n)$ coins and therefore we need a PRG with output length $m(n) = t(n)$. Let $d : \mathbb{N} \to \mathbb{N}$ be some function (which measures the stretch of the PRG). By the discussion above a PRG $G : \{0,1\}^{d(n)} \to \{0,1\}^{m(n)}$ that is pseudorandom for the family of circuits of size $m(n)^c$ for some universal constant c, can be used to get a deterministic simulation of every randomized polynomial time algorithm.[1]

On input length n, the deterministic simulation described above runs the pseudorandom generator on $2^{d(n)}$ strings and therefore its running time is roughly $2^{d(n)}$ times the running time of G. If we are shooting for a polynomial time deterministic simulation we need $d(n) = O(\log n)$ and should require that G runs in time polynomial in n. This setting of parameters is often referred to as the "high-end" as it asks for a PRG with exponential stretch $m(d) = 2^{\Omega(d)}$. At the "low-end" we consider pseudorandom generators with only polynomial stretch (namely $m(d) = d^c$ for an arbitrary constant $c > 1$). When using such PRGs we cannot get a polynomial time simulation (even if G runs in polynomial time) as going over all $2^{d(n)}$ seeds takes more than polynomial time. Still, even for this choice we can get nontrivial deterministic simulation that runs in sub-exponential time. Summing up the technical discussion we have that:

PRGs for deterministic simulation: Pseudorandom generators for polynomial size circuits can be used to produce deterministic simulation of polynomial time randomized algorithms. For this application we might as well allow the pseudorandom generator to run in time exponential in its seed length d (even if this time is super-polynomial in n). This is because the deterministic simulation runs in time exponential in the seed length anyway. Having fixed the running time of the pseudorandom generator, it is the stretch that governs the efficiency of the deterministic simulation. Exponential stretch gives polynomial time simulation, and polynomial stretch gives sub-exponential time simulation.

PRGs for reducing number of coins: If we use PRGs with polynomial stretch to reduce the number of coins used by a randomized polynomial time algorithm from $m(n)$ to $d(n)$ then we must require that G runs in time polynomial in n if we want the target randomized algorithm to run in polynomial time.

2.3 Hardness versus randomness tradeoffs

It is not hard to show that pseudorandom generators that fool polynomial size circuits give rise to circuit lower bounds. Intuitively, this is because the definition of PRGs explicitly says that there is some task that polynomial size circuit cannot perform. Proving circuit lower bounds is one of the major open problems of complexity theory and there are indications that this is a hard problem [RR97, AW08]. While the lower bounds implied by PRGs seem weaker than those needed to show NP≠P, it still seems hard to prove such bounds. Consequently, current constructions of pseudorandom generators are based on unproven circuit lower bounds. This approach is referred to

[1] The universal constant c appears because there is some blow-up when simulating algorithms by circuits. This is a minor detail that can be ignored by the reader. We remark that $c = 2$ suffices.

as the "hardness versus randomness paradigm" and was initiated by [BM84, Yao82]. A line of work initiated by [NW94] constructs pseudorandom generators using the following unproven assumption:

Assumption 1 (NW hardness assumption) *There is a Boolean function h such that:*

- *h is computable in time $2^{O(d)}$ on inputs of length d.*
- *h cannot be computed by circuits of size $m(d)$ (where $m(d)$ is some function $d < m(d) < 2^d$ that governs the strength of the assumption).*

Hardness versus randomness tradeoffs [BFNW93, IW97, STV01, SU05, Uma03] show that the assumption above can be used to construct PRGs that run in time $2^{O(d)}$, stretch $O(d)$ bits into $m(d)^{\Omega(1)}$ bits, and fool circuits of size $m(d)^{\Omega(1)}$. In words, up to the constants hidden in the $O(\cdot), \Omega(\cdot)$ above, hardness can be transformed into pseudorandomness and the strength of the lower bound translates into the stretch of the PRG. These results are the best that can be hoped for (up to the constants hidden in the $O(\cdot), \Omega(\cdot)$) in the sense that a PRG with these parameters implies the original hardness assumption (up to constants). Together with the discussion in the previous section this approach gives:

High-end derandomization: Choosing $m(d) = 2^{\Omega(d)}$, that is assuming h has "exponential hardness" gives a PRG with exponential stretch which in turn implies that $BPP = P$ [IW97].

Low-end derandomization: Choosing $m(d) = d^{\omega(1)}$, that is assuming that h has "polynomial hardness" translates into a PRG with arbitrary polynomial stretch which in turn implies a sub-exponential time deterministic simulation of BPP [BFNW93].

We remark that if the assumption holds at all then it must hold for every function complete for $E = DTIME(2^{O(n)})$, and so it is sufficient to assume that a specific function h is hard.[2] It also turns out that many of the constructions of PRGs that appear in the literature are versatile in the sense that they produce PRGs that fool any circuit class $\mathcal{C}$ when given a function that cannot be computed by the same circuit class. Corollaries are:

PRGs for constant depth circuits. There are unconditional circuit lower bounds for AC^0 (that is circuits with polynomial size and constant depth) [Hås86]. Therefore, hardness versus randomness tradeoffs give PRGs against such circuits [Nis91, NW94, Vio07]. Unfortunately, the best known lower bounds are not strong enough to produce "high-end" PRGs and therefore the resulting derandomization of randomized AC^0 takes more than polynomial time.

PRGs for nondeterministic circuits. PRGs for non-deterministic circuits can be constructed if we require that the function h cannot be computed by non-deterministic circuits [KvM02, SU06]. Such PRGs are useful for derandomizing the class AM of "Arthur-Merlin protocols" [KvM02, MV05, SU05, SU06] and are also useful in other contexts (as we see later on).

2.4 Derandomization of BPP implies circuit lower bounds

While PRGs easily imply circuit lower bounds it is not clear that PRGs are necessary for "wholesale derandomization". Nevertheless, [IKW02, KI04] show that sub-exponential time simulation of BPP

[2] Furthermore, the PRGs and resulting deterministic simulation use the function h as a "black-box". This allows us to explicitly produce a deterministic polynomial time algorithm for every randomized polynomial time algorithm and it is only the correctness of the simulation that relies on unproven assumptions.

implies certain circuit lower bounds. The circuit lower bounds implied by these works are not known to imply PRGs or derandomization. More specifically, [KI04] prove that if BPP=P then either (i) NEXP (that is non-deterministic exponential time) does not have polynomial size circuits, or (ii) The permanent function does not have polynomial size arithmetic circuits. This result shows show that circuit lower bounds are necessary for obtaining derandomization. We are however far from understanding the precise tradeoff between "hardness" and "derandomization".

3 Derandomization on inputs that are feasibly generated

In this section we explain how the notion of derandomization on inputs that are feasibly generated arises naturally from the research on always-correct derandomization. We avoid stating results formally and do not go into details. Details can be found in the survey article [Kab02].

3.1 PRGs against uniform algorithms

It is somewhat strange that a non-uniform model of computation such a polynomial size circuits appears when studying a question on the complexity classes BPP and P that consider uniform models of computation. A natural question is whether we obtain meaningful derandomization if we use PRGs that fool uniform computation such as polynomial time algorithms (rather than polynomial size circuits).

More specifically, consider a PRG with the parameters specified in Section 2 (namely, $G : \{0,1\}^{d(n)} \rightarrow \{0,1\}^{m(n)=n^{O(1)}}$ that is computable in time $2^{O(d(n))}$) that fools algorithms that run in time polynomial in n (namely, for every polynomial time algorithm D, for all but finitely many input lengths n, D cannot distinguish between $G(U_{d(n)})$ and $U_{m(n)}$). When given a polynomial time randomized algorithm A we consider the deterministic simulation B given in Section 2.2 using the PRG above. In contrast to the case of PRGs that fool circuits, it no longer follows that B simulates A correctly on every input. This is because the correctness of B on an input x relied on the guarantee that G fools the distinguisher $C_x(r) = A(x,r)$ (which can indeed be implemented by a polynomial size circuit for every choice of x). However, it could be the case that there is an infinite sequence of inputs $\{x_n\}_{n\in\mathbb{N}}$ (one for every input length n) such that the family of circuits $\{C_{x_n}\}_{n\in\mathbb{N}}$ is not implementable by a uniform polynomial time algorithm. In such a case we have no guarantee that B correctly simulates A on this sequence of inputs. Nevertheless, one could argue that failing to simulate A correctly on this sequence is not so bad as such a sequence of inputs cannot be feasibly generated by uniform computation!

It follows that using a PRG that fools uniform polynomial time algorithm we can guarantee that it is infeasible to produce inputs on which the deterministic simulation fails. More precisely we get that for every uniform polynomial time "refuting" algorithm R which generates inputs (that is, on input 1^n, R produces an input x_n of length n) it holds that B simulates A correctly on all but finitely many inputs from the sequence $\{x_n\}_{n\in\mathbb{N}}$. This follows as otherwise, the uniform polynomial time algorithm $D(r) = A(R(1^n), r)$ distinguishes the output of the PRG from uniform, for infinitely many input lengths.

3.2 Uniform hardness versus randomness tradeoffs

We have seen that a meaningful notion of derandomization is obtained if we can construct PRGs that fool uniform polynomial time algorithms. While it is unknown how to construct such PRGs unconditionally, it is not known that such PRGs imply circuit lower bounds. *Uniform* hardness

versus randomness tradeoffs are transformations which transform functions that are hard against uniform algorithms into PRGs for uniform algorithms. This is the natural analog of "standard" hardness versus randomness tradeoffs that discuss circuits. The conclusion of uniform tradeoffs is that under a hardness assumption, polynomial time randomized algorithms can be derandomized on feasibly generated inputs. In all the results in the literature it is not sufficient to assume a function that cannot be computed by deterministic algorithms and one requires a function that cannot be computed by randomized algorithms. (This is unfortunate as straightforward diagonalization often unconditionally yields the deterministic version of the hardness assumption).

The goal of this research is to achieve uniform hardness versus randomness tradeoffs matching the parameters of those achieved by non-uniform tradeoffs. A seminal result of this form is due to [IW01]. This tradeoff imitates the "low-end" non-uniform hardness versus randomness tradeoff of [BFNW93] mentioned in Section 2 when replacing "circuits" with "algorithms" in assumption 1. The obtained result is that assuming $EXP \neq BPP$ then every polynomial time randomized algorithm has a sub-exponential time deterministic algorithm such that it is infeasible to generate inputs on which the simulation fails. In fact, this holds even against *randomized* algorithms that are trying to generate inputs on which the simulation fails. This means that on every distribution that is samplable by a polynomial time algorithm, the deterministic simulation succeeds with high probability! In addition to providing meaningful derandomization under a seemingly weaker assumption, the result of [IW01] can also be interpreted as a "gap theorem" showing that unless randomized algorithms are extremely strong (and can solve every problem in EXP) then they have to be weak in the sense that they can be derandomized. We mention that achieving the "high-end" version of [IW01] is an open problem. Some progress on this problem is given in [TV07].

Obtaining uniform tradeoffs Given the aforementioned versatility of hardness versus randomness tradeoffs, it is natural to expect that these transformations yield PRGs that fool uniform algorithms when supplied with functions that cannot be computed by uniform algorithms. If this had been the case then uniform hardness versus randomness tradeoffs would have followed directly from non-uniform ones. Unfortunately, this is not the case. It turns out that all non-uniform hardness versus randomness tradeoffs in the literature are proven by *reductions* which transform a distinguisher D for the suggested PRG into a procedure that computes the suggested hard function h. These reductions have the property that one obtains a small circuit for h when starting from a small circuit D, and this is why non-uniform hardness versus randomness tradeoffs follow. Unfortunately, these reductions produce a non-uniform circuit for h even when D is a uniform algorithm. The crux of uniform hardness versus randomness tradeoffs is to design more efficient reductions that produce an efficient randomized algorithm for h when D is a uniform algorithm. Consequently, new techniques were developed to obtain uniform tradeoffs. The reader is referred to [Kab02] and to the references therein for details.

We remark that in addition to uniform hardness versus randomness tradeoffs for derandomizing BPP, there is also research on uniform hardness versus randomness tradeoffs for derandomizing the class AM of "Arthur-Merlin games" [GSTS03, SU09].

4 Typically correct derandomization

In this section we survey some recent results on typically-correct derandomization. We start with a formal definition of typically-correct derandomization.

Definition 2 (Typically-correct derandomization [GW02]) *Let $A : \{0,1\}^n \times \{0,1\}^m \to \{0,1\}$ be a randomized procedure that computes some function f. We say that a function $B : \{0,1\}^n \to \{0,1\}^m$ is a typically-correct simulation of A with error η if*

$$\Pr_{X \leftarrow U_n}[B(X) \neq f(X)] \leq \eta.$$

Let A be a randomized algorithm and $\eta : \mathbb{N} \to [0,1]$ be a function. We say that a function B is a typically-correct simulation of A with error η if for every input length n, B is a typically-correct simulation of A with error $\eta(n)$ on that input length.

4.1 Recent results on typically-correct derandomization

Typically-correct derandomization of BPP As typically-correct derandomization is a relaxation of always-correct derandomization we may hope that polynomial time typically-correct simulation of BPP can be achieved unconditionally, or at least under assumptions weaker than those used to obtain BPP=P. Current results however, rely on unproven assumptions that are incomparable to those used to obtain BPP=P. In Table 1 we list some results on always-correct and typically-correct polynomial time simulation of BPP. In the remainder of this section we sketch the proofs of some of these results and discuss the different assumptions and parameters that arise in them. Loosely speaking, the proofs of these results rely on PRGs with different parameters than those used to obtain BPP=P and this is why different kinds of assumptions come up.

Typically-correct derandomization of randomized $\mathbf{AC^0}$ Our view is that the assumptions used for typically-correct derandomization are in some sense weaker than those used for always-correct derandomization. Specifically, if we restrict our attention to constant depth circuits then these assumptions are known to hold unconditionally while the assumptions used for BPP=P do not. As a consequence, using hardness versus randomness tradeoffs with known lower bounds against AC^0, it is possible to obtain polynomial time typically-correct simulation of randomized AC^0. This is on contrast to the always-correct case in which the best known simulation takes time $2^{\mathrm{polylog}(n)}$ [Nis91]. In Table 2 we list some results on always-correct and typically-correct simulation of randomized AC^0.

Typically-correct derandomization and circuit lower bounds It is shown in [KvMS09] that typically-correct polynomial time simulation of BPP that makes very few errors (namely 2^{n^α} errors for every constant $\alpha > 0$ as guaranteed by [GW02] in Table 1) implies circuit lower bounds. This can be seen as a generalization of the results of [KI04] that shows that always-correct simulation implies circuit lower bounds.

When always-correct derandomization is impossible Typically-correct derandomization applies in settings where always-correct derandomization is known to be impossible. The first result of this kind is due to [Zim08] and gives typically-correct derandomization of randomized decision trees. Subsequent work by [Sha09] gives an alternative method that also applies to randomized communication protocols and streaming algorithms. Both approaches rely on various kinds of "randomness extractors" (see [NTS99, Sha02, Vad02] for survey articles on randomness extractors). It was later observed in [KvMS09] that the approach of [Sha09] can be recast in the terminology of "seed-extending pseudorandom generators" which we explain in Section 4.4. Moreover, [KvMS09] shows that the latter approach is more general and uses it to obtain typically-correct derandomization of randomized multiparty communication protocols.

Table 1: Polynomial time typically-correct simulation of BPP.

Reference	Hardness Assumption	Fraction of errors
[IW97]	$\mathrm{DTIME}(2^{O(n)}) \not\subseteq \mathrm{SIZE}(2^{\Omega(n)})$	0 (always-correct)
[GW02]	$\forall d > 1$, P is $\frac{1}{n}$-hard for $\mathrm{SIZE}^{\mathrm{SAT}}(n^d)$	$2^{-n} \cdot 2^{n^\alpha}$ for arbitrary $\alpha > 0$
[Sha09]	$\forall d > 1$, P is $(\frac{1}{2} - 2^{-n^{\Omega(1)}})$-hard for $\mathrm{SIZE}(n^d)$	$2^{-n^{\Omega(1)}}$
[KvMS09]	$\forall d > 1$, P is $\frac{1}{n}$-hard for $\mathrm{SIZE}(n^d)$	$\frac{1}{n^c}$ for arbitrary $c > 1$

The notation $\mathcal{D}$ is α-hard for $\mathcal{C}$ says that there is a language in $\mathcal{D}$ such that for every circuit C in class $\mathcal{C}$ and for all but finitely many input lengths n, C errs on at least an α-fraction of the inputs of length n. The class $\mathrm{SIZE}(s(n))$ refers to circuits of size $s(n)$, and $\mathrm{SIZE}^{\mathrm{SAT}}(s(n))$ refers to circuits of size $s(n)$ that have oracle access to an NP-complete language.

Table 2: Typically-correct simulation of randomized AC^0.

Reference	Randomized version of class	Simulation time	Fraction of errors
[Nis91]	Standard AC^0	$2^{\mathrm{polylog}(n)}$	0 (always-correct)
[Vio07]	AC^0 + parity gates + few arbitrary symmetric gates	sub-exponential	0 (always-correct)
[Sha09]	Standard AC^0	polynomial	$n^{-\omega(1)}$
[KvMS09]	AC^0 + parity gates + few arbitrary symmetric gates	polynomial	$n^{-\omega(1)}$

4.2 Using the input to generate random coins

All recent results on typically-correct derandomization build on the following idea suggested in [GW02]: A deterministic simulation of a randomized procedure A can try and generate "random coins" for A by applying a function on the input x that is given to A. Let us examine this idea in the naive case in which we use the *input itself* as random coins. More specifically, we say that a randomized procedure $A : \{0,1\}^n \times \{0,1\}^m \to \{0,1\}$ uses *sublinear randomness* if $m \leq n$. Note that in this case we can pretend that A uses exactly n random coins by discarding the unnecessary $n-m$ random coins. We remark that randomized algorithms that use sublinear randomness are rare exceptions in the literature on randomized algorithms and our goal is to derandomize algorithms with polynomial randomness.

Given a randomized procedure A with sublinear randomness that computes some function f we consider the deterministic procedure

$$B(x) = A(x, x).$$

This does not seem like a good idea as B may err on all inputs. This happens for example if A has the property that for every $x \in \{0,1\}^n$, every $r \neq x$ has $A(x, r) = f(x)$ while $A(x, x) \neq f(x)$. Thus, B is not necessarily a typically-correct simulation of A. Intuitively, the problem is that when choosing $x \leftarrow U_n$ we are using random coins $r = x$ that are correlated with the input x. Note that the same problem occurs if we try and produce r as a function of x. Nevertheless, some of the research we survey next achieves typically-correct derandomization of interesting randomized algorithms using this approach.

4.3 Modifying the randomized algorithm

In this section we present the approach of [GW02]. We remark that the presentation given here is different than that given in [GW02]. The high level idea is that one can avoid the bad example

above if the randomized procedure has some special properties. We say that a randomized procedure $A : \{0,1\}^n \times \{0,1\}^m \to \{0,1\}$ computes a function f with *input-oblivious* success $1-\rho$ if there exists a set $W \subseteq \{0,1\}^r$ of size $(1-\rho) \cdot 2^r$ such that for every $x \in \{0,1\}^n$ and every $r \in W$, $A(x,r) = f(x)$. Intuitively, such procedures do not mind if their coin tosses are correlated with the input. More precisely, we have that:

Proposition 3 *If A is a randomized procedure that uses sublinear randomness and computes f with input-oblivious success $1-\rho$ then the deterministic procedure $B(x) = A(x,x)$ is a typically-correct simulation of A with error ρ.*

Proof: This follows immediately as we can assume w.l.o.g. that $m = n$ and for every $x \in W$ we have that $A(x,x) = f(x)$.

The implication above does not seem interesting as we are assuming that A has very restricting properties. Namely that A uses sublinear randomness and has input-oblivious success. Our plan is to transform general polynomial time algorithms into this form. It is standard that we can transform every randomized polynomial time algorithm into one that has input oblivious success.

Proposition 4 *Every problem in BPP has a randomized polynomial time algorithm that computes it with input-oblivious success.*

Proof: Given a randomized polynomial time algorithm A we can consider a randomized algorithm A' with amplified success probability: A' simulates A using ℓ independent coin tosses $r_1, \ldots, r_\ell$ and outputs the majority vote. That is, when given a string $r' \in \{0,1\}^{m\ell}$ we view it as a concatenation $r' = (r_1, \ldots, r_\ell)$ of strings of length m and define $A'(x,r') = majority_{1 \leq i \leq \ell} A(x,r_i)$. By a standard application of the Chernoff bound we have that A' has success $1 - 2^{-\Omega(\ell)}$ for $\rho \geq 2/3$. We choose $\ell = \Theta(n)$ so that the success is at least $1 - 2^{-2n}$. This means that for every $x \in \{0,1\}^n$ the fraction of strings r' on which A' errs is at most 2^{-2n} and by a union bound A' has input-oblivious success of $1 - 2^{-n}$.

Note however, that the argument above produces a randomized algorithm A' that uses super-linear randomness even if A uses sublinear randomness. Thus, in order to obtain typically-correct simulation of A we would like to reduce the number of random coins used by A to a sublinear number. By the discussion in Section 2 we can reduce the number of random coins of a randomized algorithm using a PRG. This doesn't seem helpful since PRGs can be used to give always-correct derandomization of A and we are shooting for the weaker goal of typically-correct derandomization.

A key observation is that the parameters that are needed for our application are different than those that come up in always-correct derandomization. Specifically, it is sufficient to use a PRG with polynomial stretch $G : \{0,1\}^{d(n)=n} \to \{0,1\}^{m(n)=n^{O(1)}}$ to construct a randomized algorithm $A''(x,s) = A'(x,G(s))$ that uses sublinear randomness. We stress that as we want A'' to run in polynomial time, we must require that G runs in time polynomial in n. This is contrast to the use of PRGs with polynomial stretch for always-correct derandomization explained in Section 2.2 in which we can allow the PRG to run in time exponential in its seed length. Summing up, the discussion above we have that:

Proposition 5 *Let A' be a randomized algorithm that computes a function f with success $1-\rho$, runs in time $t(n) = poly(n)$ and uses $m(n) \leq t(n)$ random coins and let $G : \{0,1\}^{d(n)} \to \{0,1\}^{m(n)}$ be a PRG that is ϵ-pseudorandom for circuits of size $t(n)^c$ for some universal constant c and assume that G is computable in time polynomial in n. Then, $A''(x,s) = A'(x,G(s))$ is a randomized algorithm that computes f with success $1-\rho-\epsilon$.*

For our application it is not sufficient that A'' computes the same function as A'. We need A'' to have input-oblivious success. This does not seem to follow when using PRGs that fool circuits. However, it follows if we make the stronger requirement that G fools non-deterministic circuits.

Proposition 6 *In Proposition 5, assume furthermore that A' computes a function f with input-oblivious success $1-\rho$ and G is ϵ-pseudorandom for non-deterministic circuits. Then, A'' computes f with input-oblivious success $1-\rho-\epsilon$.*

Proof: Fix some input length n. By the guarantee that A' has input oblivious success we have that there exists a set $W \subseteq \{0,1\}^{m(n)}$ of size $(1-\rho)\cdot 2^{m(n)}$ such that for every $x \in \{0,1\}^n$ and $r \in W$, $A'(x,r) = f(x)$. Adleman's theorem [Adl78] says that BPP$\subseteq$P/poly and thus f can be computed by a polynomial size circuit. Moreover, the size of this circuit is bounded by $t(n)^c$ for some universal constant c. Consider the polynomial size non-deterministic circuit D which on input $r \in \{0,1\}^{m(n)}$ guesses an input $x \in \{0,1\}^n$ and accepts iff $A'(x,r) \neq C(x)$. By definition we have that D rejects every input in W and thus rejects at least a $1-\rho$ fraction of the strings of length $m(n)$. If A'' does not compute f with input-oblivious success $1-\rho-\epsilon$, then for more than a $\rho+\epsilon$ fraction of the strings s of length $d(n)$ there exist x such that $A''(x,s) \neq f(x)$ (or equivalently $D(G(s)) = 1$). In other words, D distinguishes between $G(U_{d(n)})$ and $U_{m(n)}$. (We remark that a more careful analysis could have relied on PRGs that fool uniform Merlin-Arthur protocols rather than non-deterministic circuits. The former are seemingly weaker as we know that MA$\subseteq$NP/poly).

Summing up we have that if we can construct PRGs as required in Proposition 6 with $d(n) \leq n$ then every randomized polynomial time algorithm A can be simulated by a randomized algorithm A' with input-oblivious success which in turn can be simulated by a randomized algorithm A'' with input-oblivious success and sub-linear randomness. By proposition 3 the latter has a typically-correct polynomial time deterministic simulation by simply taking $B(x) = A''(x,x)$.

The PRG that we need for this application is incomparable to PRGs that are known to imply BPP=P. It is weaker in the sense that it only has polynomial stretch (compared to exponential stretch for BPP=P). However, it is stronger in the sense that it needs to fool non-deterministic circuits (rather than deterministic ones) and that it needs to run in time polynomial in its seed length (rather than exponential in its seed length). Using hardness versus randomness tradeoffs such PRGs can be constructed from assumptions that are incomparable to those used to derive BPP=P [KvM02]. The precise assumption is given in Table 1.

In contrast to the case of always-correct derandomization (which allows the PRG to run in time exponential in its seed length) we do not know how to construct PRGs that run in polynomial time in their seed length based on worst-case assumptions. This is the reason why the assumption in Table 1 relies on functions that are hard on average. Moreover, as we want the PRG to fool non-deterministic circuits we require lower bounds against circuits that have oracle access to an NP complete language.

Optimizing the error of the typically-correct simulation using extractors We would like to get a typically-correct simulation that errs on few inputs. By proposition 3 the error of the simulation is inherited from the error of the randomized algorithm A'' which is in turn dominated by the error of the PRG. This suggests that we may want to optimize this parameter. However, it turns out that there is an easier way to control the error of the simulation. We can amplify the success probability of A'' before applying Proposition 3.

For this purpose we choose the seed length of G to be $d(n) = n^\alpha$ for some small constant $\alpha > 0$ (and note that this choice still leads to polynomial stretch). It now follows that algorithm A''

uses $d(n) = n^\alpha$ random coins. We can amplify its success probability by running it many times using "fresh random coins" (as in the proof of proposition 4). We can use $\ell = n^{1-\alpha}$ runs while still maintaining sublinear randomness. It is easy to verify that such amplification preserves input-oblivious success. This leads to error $2^{-\Omega(n^{1-\alpha})}$ which is quite good. However, we can do better by using better techniques for "deterministic amplification". Specifically, using amplification based on randomness extractors [Zuc97] we can obtain a randomized algorithm which uses n random coins and has error $2^{-n} \cdot 2^{n^\alpha}$. For this error it is more convenient to count the number of inputs on which the simulation errs rather than the fraction, and we obtain a simulation that errs on only 2^{n^α} inputs of the 2^n possible inputs of length n.

4.4 Using seed-extending pseudorandom generators

In this section we present the approach of [KvMS09]. Let $A : \{0,1\}^n \times \{0,1\}^m \to \{0,1\}$ be a randomized procedure that computes some function f with success $1-\rho$. Let $E : \{0,1\}^n \to \{0,1\}^m$ be some function and consider the deterministic procedure $B(x) = A(x, E(x))$. We are looking for sufficient conditions under which B is a typically-correct simulation of A. We know that A errs with probability less than ρ on every input and therefore it also achieves this when the input x is chosen at random. Namely,

$$\Pr_{X \leftarrow U_n, R \leftarrow U_m}[A(X,R) \neq f(X)] < \rho.$$

When using a function E to "deterministically produce random coins", the error of the simulation $B(x) = A(x, E(x))$ is given by:

$$\Pr_{X \leftarrow U_n}[A(X, E(X)) \neq f(X)].$$

Note that if the latter probability is larger than $\rho + \epsilon$ then the procedure

$$D_A(x,r) = \begin{cases} 1 & A(x,r) \neq f(x) \\ 0 & A(x,r) = f(x) \end{cases}$$

distinguishes between the distribution (X, R) and $(X, E(X))$ that appear above. This suggests the following definition.

Definition 7 *A function $E : \{0,1\}^n \to \{0,1\}^m$ is an ϵ-seed-extending PRG for a class $\mathcal{C}$ of procedures $C : \{0,1\}^n \times \{0,1\}^m \to \{0,1\}$ if for every $C \in \mathcal{C}$,*

$$|\Pr_{X \leftarrow U_n, R \leftarrow U_m}[C(X,R) = 1] - \Pr_{X \leftarrow U_n}[C(X, E(X)) = 1]| \leq \epsilon$$

The following proposition follows immediately.

Proposition 8 *Let $A : \{0,1\}^n \times \{0,1\}^m \to \{0,1\}$ be a randomized procedure that computes some function $f : \{0,1\}^n \to \{0,1\}$ with success $1-\rho$ and let E be an ϵ-seed-extending PRG for the procedure D_A defined above. Then, the deterministic procedure $B(x) = A(x, E(x))$ is a typically-correct simulation of A with error $\rho + \epsilon$.*

We now observe that if A is a polynomial time randomized algorithm then D_A can be implemented by a polynomial size circuit. This follows as by Adleman's theorem, BPP$\subseteq$P/poly and therefore f has a polynomial size circuit. Examining Adleman's proof, one can bound the size of D_A by a universal polynomial of the running time of A. Consequently, we have that:

Proposition 9 *Let A be a randomized algorithm that computes a function f with success $1-\rho$, runs in time $t(n) = poly(n)$ and uses $m(n) \leq t(n)$ random coins and let $G : \{0,1\}^{d(n)} \rightarrow \{0,1\}^{m(n)}$ be a PRG that is ϵ-pseudorandom for circuits of size $t(n)^c$ (for some universal constant $c > 1$) and assume that G is computable in time polynomial in n and has the form $G(x) = (x, E(x))$. Then, $B(x) = A(x, E(x))$ is a deterministic polynomial time typically-correct simulation of A with error $\rho + \epsilon$.*

It turns out that some hardness versus randomness tradeoffs in the literature (and in particular, the Nisan-Wigderson PRG of [NW94]) construct PRGs of the form $G(x) = (x, E(x))$. Under the right assumptions, these tradeoffs produce PRGs that fool polynomial size circuits and have the right parameters. Such PRGs are seed-extending and can be used to obtain typically-correct derandomization.

The parameters used in Proposition 9 are identical to those that come up in Proposition 5. Note that unlike the case of proposition 6 we do not require PRGs that fool non-deterministic circuits. As existing constructions of PRGs are seed-extending, proposition 9 gets typically-correct derandomization of BPP under an assumption that is weaker than that used by [GW02] in the previous section. The precise assumption is listed in Table 1.

Nevertheless, the approach of [GW02] has the advantage that it can produce a typically-correct simulation that errs on much fewer inputs. The approach described in this section produces error $\rho + \epsilon$. We can amplify the success probability of the algorithm A prior to applying the argument and can therefore reduce ρ below 2^{-n}. Thus, the error of the final simulation is inherited from the error of the PRG. In contrast to the approach of [GW02] we do not know how to further reduce the error.

Following the discussion above we would like to use PRGs with low error. It is possible to tweak the hardness assumption so that it yields PRGs with low error. However, it is impossible to achieve error that is as low as that achieved by [GW02].

5 Conclusion

Various relaxed notions of derandomization are considered in the literature. Unlike always-correct derandomization, some of these notions are not known to imply circuit lower bounds. The hope is that it may be possible to achieve such derandomization unconditionally, or at least using assumptions that are weaker than those used/required by always-correct derandomization. Current work surveyed in this article can be seen as a first step in this direction.

Acknowledgement

I am grateful to Lane Hemaspaandra for inviting me to write this survey article and for helpful comments.

References

[Adl78] L. Adleman. Two theorems on random polynomial time. In *Proceedings of the 19th Annual IEEE Symposium on Foundations of Computer Science*, 1978.

[AW08] S. Aaronson and A. Wigderson. Algebrization: a new barrier in complexity theory. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing*, pages 731–740, 2008.

[BFNW93] L. Babai, L. Fortnow, N. Nisan, and A. Wigderson. BPP has subexponential time simulations unless EXPTIME has publishable proofs. *Computational Complexity*, 3(4):307–318, 1993.

[BM84] M. Blum and S. Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM Journal on Computing*, 13(4):850–864, November 1984.

[GSTS03] D. Gutfreund, R. Shaltiel, and A. Ta-Shma. Uniform hardness versus randomness tradeoffs for Arthur-Merlin games. *Computational Complexity*, 12(3-4):85–130, 2003.

[GW02] O. Goldreich and A. Wigderson. Derandomization that is rarely wrong from short advice that is typically good. In *Randomization and Approximation Techniques, 6th International Workshop, RANDOM 2002, Cambridge, MA, USA*, pages 209–223, 2002.

[Hås86] J. Håstad. Almost optimal lower bounds for small depth circuits. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, pages 6–20, 1986.

[IKW02] R. Impagliazzo, V. Kabanets, and A. Wigderson. In search of an easy witness: exponential time vs. probabilistic polynomial time. *J. Comput. Syst. Sci.*, 65(4):672–694, 2002.

[Imp06] R. Impagliazzo. Can every randomized algorithm be derandomized? In *Proceedings of the 38th Annual ACM Symposium on Theory of Computing*, pages 373–374, 2006.

[IW97] R. Impagliazzo and A. Wigderson. $P = BPP$ if E requires exponential circuits: Derandomizing the XOR lemma. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 220–229, 1997.

[IW01] R. Impagliazzo and A. Wigderson. Randomness vs time: Derandomization under a uniform assumption. *J. Comput. Syst. Sci.*, 63(4):672–688, 2001.

[Kab01] V. Kabanets. Easiness assumptions and hardness tests: Trading time for zero error. *J. Comput. Syst. Sci.*, 63(2):236–252, 2001.

[Kab02] V. Kabanets. Derandomization: A brief overview. In *Electronic Colloquium on Computational Complexity, technical reports, TR 02-008*, 2002.

[KI04] V. Kabanets and R. Impagliazzo. Derandomizing polynomial identity tests means proving circuit lower bounds. *Computational Complexity*, 13(1-2):1–46, 2004.

[KvM02] A. Klivans and D. van Melkebeek. Graph nonisomorphism has subexponential size proofs unless the polynomial-time hierarchy collapses. *SIAM J. Comput.*, 31(5):1501–1526, 2002.

[KvMS09] J. Kinne, D. van Melkebeek, and R. Shaltiel. Pseudorandom generators and typically-correct derandomization. In *APPROX-RANDOM*, volume 5687 of *Lecture Notes in Computer Science*, pages 574–587. Springer, 2009.

[Lev86] L. A. Levin. Average case complete problems. *SIAM J. Comput.*, 15(1):285–286, 1986.

[Mil01] P. Bro Miltersen. Derandomizing complexity classes. In *Handbook of Randomized Computing, Kluwer*, pages 843–941, 2001.

[MV05] P. Bro Miltersen and N. V. Vinodchandran. Derandomizing Arthur-Merlin games using hitting sets. *Computational Complexity*, 14(3):256–279, 2005.

[Nis91] N. Nisan. Pseudorandom bits for constant depth circuits. *Combinatorica*, 11(1):63–70, 1991.

[NTS99] Noam Nisan and Amnon Ta-Shma. Extracting randomness: A survey and new constructions. *J. Comput. Syst. Sci.*, 58(1):148–173, 1999.

[NW94] N. Nisan and A. Wigderson. Hardness vs. randomness. *Journal of Computer and System Sciences*, 49(2):149–167, October 1994.

[RR97] A. R. Razborov and S. Rudich. Natural proofs. *Journal of Computer and System Science*, 55(1):24–35, 1997.

[Sha02] R. Shaltiel. Recent developments in explicit constructions of extractors. *Bulletin of the EATCS*, 77:67–95, 2002.

[Sha09] R. Shaltiel. Weak derandomization of weak algorithms: Explicit versions of Yao's Lemma. In *IEEE Conference on Computational Complexity*, pages 114–125. IEEE Computer Society, 2009.

[STV01] M. Sudan, L. Trevisan, and S. P. Vadhan. Pseudorandom generators without the XOR Lemma. *J. Comput. Syst. Sci.*, 62(2):236–266, 2001.

[SU05] R. Shaltiel and C. Umans. Simple extractors for all min-entropies and a new pseudorandom generator. *J. ACM*, 52(2):172–216, 2005.

[SU06] R. Shaltiel and C. Umans. Pseudorandomness for approximate counting and sampling. *Computational Complexity*, 15(4):298–341, 2006.

[SU09] R. Shaltiel and C. Umans. Low-end uniform hardness versus randomness tradeoffs for am. *SIAM J. Comput.*, 39(3):1006–1037, 2009.

[TV07] L. Trevisan and S. P. Vadhan. Pseudorandomness and average-case complexity via uniform reductions. *Computational Complexity*, 16(4):331–364, 2007.

[Uma03] C. Umans. Pseudo-random generators for all hardnesses. *J. Comput. Syst. Sci.*, 67(2):419–440, 2003.

[Vad02] S. P. Vadhan. Randomness extractors and their many guises. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, pages 9–12, 2002.

[Vio07] E. Viola. Pseudorandom bits for constant-depth circuits with few arbitrary symmetric gates. *SIAM J. Comput.*, 36(5):1387–1403, 2007.

[Yao82] A. Chi-Chih Yao. Theory and applications of trapdoor functions. In *23rd Annual Symposium on Foundations of Computer Science*, pages 80–91, 1982.

[Zim08] M. Zimand. Exposure-resilient extractors and the derandomization of probabilistic sublinear time. *Computational Complexity*, 17(2):220–253, 2008.

[Zuc97] D. Zuckerman. Randomness-optimal oblivious sampling. *Random Struct. Algorithms*, 11(4):345–367, 1997.